目錄

《Node.js区块链开发》

【注】基于ES6的最新版，正在筹划中。

特别说明



亿书，让有知识的人富起来。我们的产品，将全部基于P2P网络进行开发设计，面向未来进行技术研发。

纸质书籍：https://item.jd.com/12206128.html

在线培训：https://www.shiguangkey.com/course/1929

联系方式，请将您的简历发送到： hr@ebookchain.org

关于（About）

本书可以作为Node.js开发加密货币的入门书籍（正式出版可能使用《Node.js区块链开发》），也可以作为亿书（及以Crypti为核心的应用Lisk）的官方开发文档。

本书分享的源码是Ebookcoin（亿书币），是亿书的强大动力。与Lisk一样（成功众筹500多万美元），都是Crypti（已经不再维护）的一个独立分支，类似于以太坊具有侧链功能，可以承载多种去中心化的应用。因此，无论您是研究**Crypti**、**Lisk**，还是**Ebookcoin**，或者学习**Node.js**前后端开发技术，本书都值得参考。

亿书的目标是打造人人可用的去中心化软件，以加密货币为驱动，促进人类知识分享。与其他竞争币不同，我们以提供落地可用的软件为核心，力争成为人类第一个“零门槛”的加密货币产品。更多详情，请看关于亿书

本书写作，也是亿书的最佳实践，从写到发布，简单、快捷。文章暂时在github上免费发布，永久免费访问地址: http://bitcoin-on-nodejs.ebookchain.org

注: 本书永久 `免费` 访问地址: http://bitcoin-on-nodejs.ebookchain.org

亿书官网： http://ebookchain.org

官方开发交流QQ群：185046161

日志（Log）

- [x] 2016-10-15 收录巴比特创始人兼CEO @长铗 的荐序。
- [x] 2016-10-14 整体校对交稿，交付印刷
- [x] 2016-10-12 完成第30篇: 完成致谢
- [x] 2016-10-10 完成第29篇: 完成后记
- [x] 2016-10-08 完成第28篇: 生产环境下的pm2部署，补充部分名词解释
- [x] 2016-10-06 完成第27篇: 测试
- [x] 2016-09-21 完成第26篇: 区块链架构设计简介
- [x] 2016-09-14 发布第25篇: 开发通用的HTML组件
- [x] 2016-09-02 发布第24篇: 轻松从Js文件生成UML类图
- [x] 2016-08-18 发布第23篇: 学点函数式编程
- [x] 2016-08-10 发布第22篇: DPoS机制
- [x] 2016-07-28 发布第21篇: 区块链
- [x] 2016-07-14 发布第20篇: 自序
- [x] 2016-07-08 发布第19篇: 交易
- [x] 2016-07-02 发布第18篇: 关于时间戳及相关问题（优化补充）
- [x] 2016-06-27 发布第17篇: 签名和多重签名
- [x] 2016-06-23 发布第16篇: 地址
- [x] 2016-06-06 发布第15篇: 共识机制，可编程的利益转移规则
- [x] 2016-05-29 发布第14篇: 利益，魔鬼与天使的共同目标
- [x] 2016-05-23 完成第13篇: 加密货币就是货币
- [x] 2016-04-28 `书链` 更名为 `亿书` ，撰写第13篇
- [x] 2016-04-17 发布第12篇 ：Ember深“坑”浅出
- [x] 2016-03-26 发布第11篇 ：一张图学会使用Async组件进行异步流程控制
- [x] 2016-03-17 发布第10篇
- [x] 2016-03-10 发布第9篇
- [x] 完成1-8篇

使用（Usage)

目录由命令行工具 gitbook-summary 自动生成。自由写作、发布，搭建自出版平台的方法，请点击这里

简要介绍如下：

(1)克隆源文

```
$ git clone https://github.com/imfly/bitcoin-on-nodejs.git
```

(2)安装gitbook

```
$ npm install -g gitbook-cli
```

(3)安装依赖包

```
cd bitcoin-on-nodejs
npm install
gitbook install
```

(4)写作构建

写作，并开启服务（构建）

```
$ gitbook serve
```

通过 `http://localhost:4000` 实时浏览

(5)生成目录

只要修改了文章标题和文件夹，就应该重新生成目录文件

```
$ npm run summary
```

(6)一键发布

```
$ npm run deploy
```

以后，只要4-6的过程就是了。

反馈（Feedback）

随时告诉我您的阅读体验和问题，也可以直接fork修改，提交PR。

贡献者（Contributors）

@imfly @Tailor @火鼎 @珍惜 @一 @Mojie @cyio @zbinlin

协议

作者

微信：kubying

前言：亿书和数字出版新时代

恰在本身内容撰写过半，与出版社签订出版合同的时候，巴比特论坛和比特时代组织了一场以“我和时代的故事”为主题的征文活动。一语双关的活动主题，响亮而鲜明，我被狠狠地触动了一下。于是，打开电脑，写下这些回忆，权当给本书加上序言。

我和我的比特时代

我钟爱编程，近乎狂热，就像别人爱好足球一样，了解我的朋友都知道这一点。2009年，刚参加工作几年，一切还算顺利。有一天，接到一位留学美国的同学一封邮件，附件里有一篇英文文档，他问我感不感兴趣。那时候，还没有比特币的概念，也不懂得一个软件产品还要弄个什么技术白皮书，至少在中国不需要，再看看满篇的新概念和数理逻辑，我直接无视了，说实话，看不懂内容，想不到它的未来。不懂的东西，又没有探索的动力，哪怕未来多么有价值，也与你无缘。

转眼到了2013年，比特币发展迅猛，价格也是高歌猛进，政府的打压、媒体的质疑，让这个原本是少数极客把玩的东西，一下子成了“网红”。“你看现在的年轻人多厉害，电脑上弄一串数字，就能卖好几千”，一位开出租的司机师傅谈到他的辛苦收入时，这样跟我感慨。当时，我好像被电了一下，但是因为家里的变故，始终沉浸在悲痛中无法自拔，被触动的亮光很快就熄灭了。再有价值的东西，与亲情、与生命相比又算得了什么呢，当心静下来，一切都不重要。

2014年初，时间终于把我叫醒。我感觉自己沉睡了很久，重新审视浑浑噩噩的这两年，突然觉得有个所谓的比特币自己可能已经错过了好多次。于是，走进网络，开始如饥似渴的补充知识，这一过又是一年多。这一年多来，在别人打球、垒长城的时候，我都沉浸在自己的代码海洋里，只不过研究的方向变了，一切都围绕着去中心化的应用开发和实践。我也不断的“买买提”，成为包括比特时代在内的几个交易网站的常客。当然，我没有一夜暴富的梦想，也没有短线操盘的空闲，只是坚信当前仍是加密货币的初期，只要踏踏实实做事，任何团队、任何产品都有机会，这是真正做过产品的人都明白的道理。

但是，谁去考察开发者的初心呢，谁又知道某些人是出于什么目的呢？看看那些时不时曝出的传销币、跑路男，在利益面前，任何人都是渺小和无力的。这也是比特币火爆的根源，用编程的方式，固化了“利益”转移的规则，实现了公开、透明、可追溯的信用体系，让比特币变成了一个自我完善的经济系统。因为体制原因，我以前从来不会在网上使用自己的真实信息，但是后来发现这不是什么好习惯。至少，在这个快速的信息时代，我们所做的很多工作都淹没在浩瀚的网络海洋里了。于是，我决心撕下伪装，学习比特币的理念，用有限的手段积累个人信用。先后注销了各种马甲，在github和巴比特上注册了自己常用的帐号，贴上自己的真实头像，像区块链那样，在网络上宣示自己的存在。这个时间点，大概就是2015年的5月份。

2016年，经历过很多小伙伴都经历的故事和两年的沉淀之后，也看清了币圈的混乱。特别是听到很多人骂，中国的技术总是不如别人的时候，心里多少有点伤感。于是，决心自己先行动起来，为改变这个现状做点事情。元旦过后的第一天，在巴比特论坛上写下了自己的心愿单（见参考），晚上发布了《Nodejs开发加密货币》开栏寄语。自那以后，除特殊情况外，平均每周完成一篇，本文算是本专栏正式发布的第20篇（电子书更多）。半年来，一直有一种非常新奇的体验，最初仅仅是我一个人在唱独角戏，后来就有很多小伙伴看到我的文章进来，大家在一起自然会有更多新奇的想法，反过来推着我去作出进一步改进，如此反复。截止目前，仅电子书，单日的浏览量好几百，仍在持续增加中。

至此，我的比特时代正式启幕。比起那段黑暗的时光，我现在每天都被梦想叫醒，每天都在享受生命。

亿书和数字出版新时代

“每一件与众不同的绝世好东西，其实都是以无比寂寞的勤奋为前提的，要么是血，要么是汗，要么是大把大把的曼妙青春好时光”。这是一位文笔非常好的朋友大学毕业后，在QQ上给我的留言。后来，我把它设置成了个人签名，每每读到这句话，我总能被感动。后来，与那位朋友聊天，我说你真有才，能把一句话说到人的内心深处，我要不是男人，一定美美地哭一场。他一听，十分感慨，告诉我他也是抄来的，他还没有修养到可以讲出这种意味深刻的话来，也只有经历过，才能被感动。

是的，美好的东西，任何人都能感受到，哪怕是田地里的庄稼都能。我是农村出来的，爸妈从来没有读过书，地地道道的农民。爸妈做任何事情，都要经历漫长艰辛的过程，因为他们不懂设计和规划，没有专业知识，不能上网查资料，一切都要靠自己摸索。但是，在我看来，他们做的每一件事，给我相同的时间，我也不一定做得比他们更好。我妈曾经告诉我，做事得用心，用不用心，每颗庄稼都知道。这样的教育，对我的影响很大，直到现在，我都认为自己其实就是一个农民，只不过干活的那个锄头在心里，表面上换成了键盘。

工匠精神，人人推崇，但不是人人都能做到，环境往往起到很大的作用。还是我上面那位朋友，刚毕业的时候，非常风光，去了某机关报社，最初当记者，待遇优厚，特权多多，后来做编辑，自在逍遥。这中间，还经常写写书，一年下来，仅仅版税收入也非常可观。但是，最近几年好像不太好了，有一次聊天明显感觉到他的消极，问他怎么了，他说都是我们这些会编程的人闹的，互联网抢了他们的饭碗。我说，那你就抢回来吧，化敌为友，借助互联网挣更多钱。他说试过了，没那么简单，几大文学网站，那么多作者，真正挣到钱的没有几个。偶尔火起来了，大部分都被平台克扣去了，甚至连版权都得不到。加之盗版猖獗，基本入不敷出。

听着他的话，我始终沉默，当初我也想不到出路，时代变迁，谁都无法阻拦。但是，这两年我有了方向，找到了可以彻底解决的办法，这也是上面我如饥似渴学习比特币的内在动力，也是坚持分享这本书的内在动力。我并没有告诉他我的想法，因为我没有充当救世主的怪癖，只是身边人遇到问题，你有机会帮他解决而做不到时，就会成为心结。软件是给人用

的，开发者首先想到的自然是身边人，这是再正常不过的道理。有人看看别人需要什么，就能总结出一个好的项目，着实令人佩服，我自知不具备这样的素质，一点都没有。这就是亿书诞生的初衷，没有任何离奇故事，都是满满的生活小节，这多少也有点工匠精神的情结吧。

当主动往版权保护和写作分享发力的时候，一切资源就源源不断的走到你的面前。亿书，这个去中心化的版权保护和自出版平台的操作过程，大致是这样的，2015年年底，在经过一段时间的技术探索之后，我把关于打造电子书版权保护项目的想法借助巴比特论坛和盘托出，证明了想法的可行性，也吸引了很多小伙伴关注。接着，我用行动实践电子书写作和分享的全过程，并把亿书这个项目的真实源码作为本书分享的重要内容，边分享边开发，边打造开源团队，这中间提交了除亿书币源码之外的官方网站、编程语言调查项目、电子书目录生成工具以及多个前端插件，搭建了官网博客、区块链俱乐部。读者越来越多，团队日益壮大，仍有很多小伙伴在了解、考察和熟悉中。再接着，等亿书正式发布，将用亿书（软件）来继续分享亿书（源码）了。这就像C/C++这样的编程语言可以用来开发自己，开发者也是使用者，自身不断循环完善。

亿书源码来自于早期的Crypti，与Lisk（成功众筹650万美元）几乎同时建立版本库，但是二者选择了完全不同的发展路线。在钱和产品的选择上，亿书团队选择产品，钱仅是副产品。亿书定位在协同创作和版权保护，以及以此为基础的衍生品之上，并通过这种方式建设覆盖全球的P2P网络。提供侧链扩展开发能力，基于亿书开发任何类型的去中心化应用，比如：电子商务、直播平台等等。这仍然是从基础需求起步，步步搭建积木的思路和过程，与我通过写作本书，体验产品本身，然后发展亿书的思路一样，与我扛着锄头下农田干活，修理一颗一颗的秧苗一样，只不过时代变了，现在用上了机械化，玩起了去中心化而已。这是工程学的基础，再厉害的天才也无法逾越。

亿书，注定要开启数字出版新时代。

imfly 2016年7月15日 于北京

关于本书

为什么要写这本书

随着比特币的火热，区块链正在成为比肩大数据、VR、机器学习等目前炙手可热的技术之一，世界各国政府、国际性大公司纷纷布局，试图把区块链用到实际中去。但是，区块链开发技术门槛高，可供学习参考的资料少，目前市场上的书籍大多是理论介绍性的，涉及到开发的书籍也多以简单的Api调用为主，没有一本深入区块链技术核心，从代码深度去完整讲述如何开发设计区块链产品的。本书弥补了这个空白，基于一个实际运行的区块链产品，从概念到实现，完整呈现了它的开发过程。

同时，就Node.js开发类书籍而言，市面上各种档次的书籍不在少数，但是多数使用假想的例子，应付了事，对真正的开发应用，没有太多的参考指导价值。本书，另辟蹊径，通过分享上述区块链产品的设计思路、设计方法，阅读它的源码设计，穿插讲述用到的相关知识，不仅整个过程都是实例代码，配合个别章节还开发了多个可独立运行的小应用。可以说，书中每一行代码都是实际在使用的代码，具备很强的实践参考价值。另外，本书也弥补了使用Node.js开发区块链应用这个书籍市场空白。

《Node.js开发加密货币》是本什么样的书？

亿书是完全开放开源的项目（官网见链接），一个完整的类比特币的区块链产品。本书基于该项目，完全以实用为目的，把开发实践贯穿始终，内容涉及到开发区块链产品前端、后台和桌面应用的全过程。是用开发的思维反复迭代的书籍，由浅入深，详细介绍了区块链技术相关理论知识、Node.js前后台开发基础知识、加密签名技术、P2P网络实现、共识算法等，能帮助初学者快速学习入门区块链技术，深入掌握Node.js编程开发技术，帮助区块链技术从业者、Web开发者更深刻的理解相关概念和技术实现。

- 想找如何开发一款真正的区块链产品（不单单是调用某款加密货币Api）的书籍吗？这是目前世界上第一本，也是唯一的一本；
- 想找Node.js大型实践项目的书籍吗？这可能是世界上少有的一本，也可能找不到第二本；
- 想找亿书、Lisk等基于Crypti的分支项目的详细开发文档吗？这一定是世界上唯一的一本；
- 想深刻了解区块链的技术实现吗？看看本书，对于掌握区块链、共识机制等各种概念更加透彻；
- 想从事区块链(无论比特币还是其他各类竞争币)的开发吗，Node.js您一定无法逾越，这本书也必然无法错过；
- 想了解比特币原理吗？这本书不仅告诉你是什么，还从技术角度告诉你为什么，无论你

是技术还是管理，都值得参考。

怎么阅读本书

本书力图用最少的篇幅表述更丰富的内容，共分为五个部分20多个章节：

第一部分：了解加密货币，共4章。详细讲述了加密货币的相关概念，用独特的技术视角，把加密货币的基本技术要素串联起来，同时在文中自然引导读者跳转阅读下面各个部分，实现理论到实践的过度。

第二部分：Node.js入门指南，共4章。详细介绍了Node.js入门知识，并通过一个具体项目，完成对Node.js在区块链技术领域的调查和描述，整个章节也是项目架构设计必备的调研和技术选型阶段，是本书第一个完整的实践范例。

第三部分：源码解读，共9章。从架构设计的角度，层层剖析区块链的设计原理，深刻解读相关概念和技术。从项目设计的角度谋篇，第9章，详细介绍了亿书白皮书的核心内容，明确了项目的需求，教会读者如何着手研究区块链产品；第10章，从项目入口程序出发，介绍了亿书项目的整体结构；第11-17章，分别介绍了P2P网络、加密解密、签名和多重签名、区块链、共识机制等区块链核心内容及其代码实现。

第四部分：开发实践，共12章。主要是对第二和第三部分的有益补充，把在这两个部分出现的技术难点抽取出来，集中介绍。仍然以亿书项目中涉及到的实际项目为主，包含多个完整独立的小项目。第18章，总结了aysnc的用法，解决了Node.js回调流程控制问题；第19章，介绍了命令行工具的开发（含开源实例）；第20-21章，介绍了亿书官方网站的开发，对市面上的静态网站进行了总结，通过两个实例详细介绍了客户端的开发设计；第22章，详细介绍了加密解密技术；第23章，介绍了测试技术;第24-25章，介绍了部署方案；第26-27，介绍了时间戳、数据计算等更加细致的优化内容;第28-29章，主要介绍了函数式编程等编程方法论，帮助读者从更深层次写代码（含实例）。

第五部分：附录。汇总了区块链的相关概念、常见词汇的中英文对照，以及代码规范等其他内容。

参考

亿书官网：http://ebookchain.org

2016年上班第一天，留个记号，许个心愿

开栏寄语

中国的币圈发展有点不太均衡，投机心态和快餐思维，左右着币圈和中国区块链技术发展，在加密货币的整个生态链里，没有占据有利地位。

比特币的真正价值何在？我曾经写过一篇文章 点击这里，回顾和总结了个人看法。

用易经的哲学思维，从整体上来说，一件事情已然发生，必然会继续发展下去。所谓“一生二，二生三，三生万物”，一个比特币出现了，必然会出现第二个、第三个......这是世界万物发展的普遍规律，谁也无法阻止。其强大的生命力已经初露端倪。

从阶段性判断，比特币仅仅是个开始，而且是刚刚开始。当前各种环境纷繁复杂，骗局、跑路，一个个牛x的新思路，各种声音杂陈，唱涨唱衰参半。但除了比特币，没有一个像样的应用落地，所有的应用，都那么高大上，就像最初的BP机、大哥大或智能手机，只能作为少部分人的玩具和试验品。

去中心化，是人类社会的基本形态。之所以被提出来，不是这个技术怎么样先进，而是现有的技术多么的落后。p2p、加密和分布式等技术的出现，都是人类在某个阶段解决特定问题设计出来的，组合在一起成为加密货币，更接近了人类本源。

一切违背人性的东西，都要被颠覆。每一个人，都不想被限制。把你的命运，交到第三方手里，任由他摆布，你必不乐意。所以，以服务器为中心的各类有中心的应用都将逐步走向终结，仅是时间问题。

比特币的真正价值是它给了我们进入新世界的一个样板。互联网3.0时代已经来临，我们将藉此韬光养晦，脚踏实地，好好打磨自己的技术、开发落地的产品和集聚爱好这一切的人脉。我们决定做那一批踏踏实实玩技术的人。

开启这个专栏，主要有这样**3**个目的：

1.娱乐自己。我总觉得做自己不喜欢的事情，就是在浪费生命。“千金难买好心情”，比特币再有价值，也仅是一个过程和工具。

2.结识朋友。先把自个脱光了，赤裸裸的展示给别人看，真正道同的朋友会找到你，节省了很多沟通的时间和精力。

3.积累技术。算是抛砖引玉，原本加密货币是技术性很前沿的技术，放弃技术研发就是舍本逐末，丢掉大好机遇。

主要方法和步骤也有**3**个：

1.研读源码。这是最高效的学习方式，一方面会让我们快速学习node.js编程方法，另一方面会让我们深入理解区块链核心技术，再者给了我们开发的参考路线图，在技术选型、开发逻辑等方面有了参考案例，少走很多弯路，一举多得。

2.总结提升。一个项目的源码读完，自然会产生很多想法，可不可行，有没有价值，总要动手试试，这样一个试错的过程，才是知识吸收和转化的过程，读代码是基础，而这个自我总结和提升的过程同样必不可少。

3.项目实践。技术是拿来用的，还是一边学，一边开发比较好。或者参与一个项目，或者开启一个新的创意项目。

时间安排上，我会力争一周更新一篇（特殊情况除外），一篇解决一个问题。

面向的读者对象，主要是有一定编程基础的人，当然对于node.js而言，你只要写过网页的javascript，了解服务器开发基本流程就能够理解（当然，距离开发还有一段路要走）。而对于区块链技术，我们了解的可能都差不多，不懂的就在社区里问问吧。

我不太擅长写作（脑子里把写作当编程），平日的时间也不是很多，技术水平当然也非常有限，所以敢于出来班门弄斧已经是鼓足了十分的勇气。如果万一不小心坚持了下来，那必定是一个奇迹。因此，无论鼓励也好，批评也罢，都将是我前进的动力。

第一部分：了解加密货币

这部分主要针对没有接触过区块链技术的初学者。当然，对于不了解技术，在币圈混迹多年的小伙伴，也是有用的，可以帮助您从技术实现角度，更好的理解区块链的有关概念。

这类文章，我把它称作技术类软文，讲理论多一些。我们知道，对于咱们普通老百姓而言，人类语言的力量往往非常苍白，特别是在描述复杂的区块链产品的时候，远不如计算机语言简洁、明了和严谨。所以，阅读这类文章，要保持良好心态，寻求文章里有价值的东西，避免把自己的情绪与好恶掺杂其中（当然也不要受作者的情绪与好恶左右），这样才能真正有收获。

比如，在《共识机制，可编程的“利益”转移规则》里，我指出了各种共识机制的缺点，还说它们都有失败的可能，有些小伙伴就非常受不了，发应强烈，这种连一个“不”字都听不得的大有人在，明显就是害怕失败，担心损失，自欺欺人。

这几篇文章，核心就一个主题，告诉您区块链产品——加密货币就是货币，但是解释起来却是大费周章，原因是时刻避免使用太多的专业术语，还要讲透原理和内容，对于我这样的编程人员来说，有点分裂。

利益，魔鬼与天使的共同目标

前言

上篇文章《加密货币就是货币》吸引了很多关注，说明很多小伙伴因为对加密货币不了解（或者有误解），所以才会敬而远之，错失良机。

这篇文章，继续上一篇，仍然通过直白的语言来讲解技术。涉及到的内容包括未来趋势，应用场景和风险提示，让我们更直观地理解币圈里的一些概念，比如：智能合约等。

利益，主宰着人类行为

人活着到底是为了什么？我们每个人可能都问过自己这个问题。我们有时候踌躇满志，想要拥有一切。有时候又高尚地低下头，崇尚与世无争，无忧无虑。但在纷繁复杂的真实世界里，我们总会被某个力量牵引着，挣脱不开，欲罢不能。

这个力量，就是追求利益的欲望。利益，是什么？从网上查到的解释是：

> 利益是指人类用来满足自身欲望的一系列物质和精神需求，包括：金钱、权势、色欲、名声、地位等，但凡能满足人类欲望的事物， 均可称为利益。利益依附欲望而生，而人的基因确定了欲望的存在，组成社会的基本元素是人，就不可避免地出现了：阶级、政治、 战争……利益冲突决定着一切。

人们对利益的追求来源于人的本性。人具有三种本性，即求生的第一本性，懒惰的第二本性和不满足的第三本性（这是人和动物的根本区别）。所以，人们的利益也可以分为三类，即求生的利益，懒惰的利益以及不满足的利益。总之，人类欲望无止境。

历史名人，对于利益的名言，也是值得学习和思考的。马克思说过：“人们奋斗所争取的一切，都同他们的利益有关”，列宁也说：“几何公理要是触犯了人们的利益，那也一定会被推翻的”，霍尔巴赫的话更加直白：“利益是人类行动的一切动力”。

所以，我们不仅不用避讳谈利益，而且最好把利益作为我们分析和思考产品开发设计的根本因素。如此以来，对于理解人们为何对加密货币趋之若鹜就自然轻松多了。

可编程的“利益”转移手段

最近不止一次谈到“信用”这个话题，每一次都会有更加深刻的理解。第一次明确的谈信用，起源于一个公司反复的声明一件事情，但是除了声明，也没有更好的办法向公众表明自己，反而越声明越被动。我于是写了一篇文章 《请尽早把你的信誉区块链化》(见参考) ，宗旨是学

习区块链的处理方式积累个人或公司信用。

今天再次提及这个话题，原因是我在写下前言部分的时候，突然发觉单纯的讲未来趋势如何，没有任何说服力，其中缺少潜在的源动力。另外，之前，在文章前面，我经常写下这样一段话:

> 发布本文时，比特币价格xxxx元（xxx美元）。为什么一个凭空设计出来的加密货币如此受追捧？为什么微软、IBM等巨头纷纷进入？ 为什么尝试了解比特币的技术人员，都会被深深吸引？它到底有什么诱人之处？《Node.js开发加密货币》，正尝试回答这些问题。

可当时，没想过如何正面回答为什么。今天发现，在这里回答，应该是最好的时候。

（1）信用是交易的基础

我们什么时候才会讲到“信用”？什么时候才会考查他人的“信用”？很显然，就是准备与他人进行“交易”的时候。可以说，信用是交易的前提，没有信用就没有交易，有了信用才有可能交易。

经济社会，就是一个信用社会。信用不在，一切都将回归原始社会：物物交换，甚至是自给自足。因为交易风险太大，人类保护自己利益（求生的本性）和追求自己利益最大化（不满足的本性）的欲望，会促使人们放弃交易。信用，对于个人和企业而言，同等重要。

（2）信用是积累的过程

信用的建立，不是一朝一夕的事情。如果深入思考，安全、公开、可追溯，才是人类建立信任的基本要素。没有一个人一见面就说我信任你的，也没有一个人躲躲藏藏还能赢得他人信任的，一般都是“路遥知马力，日久见人心”的事情。

这里隐含的意思就是，每个人都会对另一个人的信用，在自己心里建立一个“区块链”。一旦有交易往来，就会自觉回忆过往的点点滴滴（追溯），对当下的交易进行风险评估。如果是重大交易，这个人可能还要求教第三方，通过他人的“区块链”进一步认证。

（3）信用的本质是解决了信息不对称

经济学建立在一个基本假设之上，那就是“人，是理性自私的”，也就是说一切经济行为都会朝着“利益最大化”的方向发展。人们没有办法一开始就信任你，是因为彼此“信息不对称”，没有足够信息证明自己不会在你“利益最大化”的过程中“被牺牲掉”。

“信息不对称”表现在三个方面：一个是，我的信息，你不知道;另一个是，我的信息，你知道但无法识别;第三个是，我的信息，你知道，也可以辨别，但是你无法控制。

一些经济学家对此进行了大量研究，他们认为，信息不对称造成了市场交易双方的利益失衡，影响社会的公平、公正的原则以及市场配置资源的效率，并且提出了种种解决的办法，但是仍然无法彻底解决这个问题。

直到比特币的出现，人们才真正找到了一条切实可行的方法途径。为什么说比特币会火，人人都觉得它有价值？是因为它绝对可信（至少目前是）！为什么它绝对可信，是因为区块链、加密技术和分布式网络的组合，让交易变得安全、公开、可追溯。（基本原理可以看上一篇文章分析）

所以，区块链承载的就是信用，是一个无需政府、银行或财团抵押担保的信用。如果，没有政府保障，法币将一文不值。如果没有区块链，比特币也将一文不值。这就是区块链的价值所在，也是比特币和那些真正的加密货币会火的真正原因。

换句话说，纯粹编程实现的加密货币可以让“利益”按照设定的规则转移（交易），而人类尊重这个规则就会受益，背离将毫无所获，最终的结局是人类纷纷参与其中，让这个盘子越来越大。现在你看到的比特币，就是这个结果，社会各界都预测比特币将死，可结果却是更加红火。

未来趋势

比特币之前，人类从来无法完全控制“利益”走向，比特币之后，人们终于可以对“利益”转移进行编程处理，这将给人类未来带来无限的可能。

去中心化，人类社会的基本形态，是个体交换的基本前提。之所以被提出来，不是这个技术怎么样先进，而是现有的技术多么的落后。p2p、加密解密和分布式等技术的出现，都是人类在某个阶段解决特定问题设计出来的，组合在一起成为加密货币，更接近了人类本源。

一切违背人性的东西，都要被颠覆。每一个人，都不想被限制。把你的命运，交到第三方手里，任由他摆布，你必不乐意。所以，以服务器为中心的各类有中心的应用都将逐步走向终结，仅是时间问题。

那些去中心化的应用，才是真正的符合人性的应用，必将占据主流，抢占先机。至于，是不是比特币或其他的加密货币，根本不重要，只要能让你无障碍的进行交换（人类处处不交换，交换信息、金钱、财富）。

比特币给了我们进入新世界的一个样板。全新的互联网时代已经来临，只是处在初级阶段，那些真正落地，被大众接受的应用还不存在。目前，所有的应用，仍然那么高大上，就像最初的BP机、大哥大或智能手机，只能作为少部分人的玩具和试验品。但是，很快，它将遍布这个世界。

应用场景

我们的目标是利用区块链技术，开发应用产品，所以了解它的应用场景将对我们有很大帮助。汇总一下，这项技术可以应用在这样几个方面：

（1）转账支付

这个应该是最基本的功能，很多加密货币，比如比特币，本身的目标就是实现没有第三方的直接支付，所以应用于转账、支付、借贷，都是简单和轻松的事情。特别是，加密货币之间可以直接互相兑换之后，未来或许将再也见不到有独立第三方存在的支付机构了。

（2）资金结算

区块链本身的结构体系，就是一个很好的结算系统，由于网络遍布世界各地，跨境支付与结算也是非常容易的事情。当前的银行结算体系非常复杂，每个机构都有自己的记账，结算难度大，周期长。于是有人，联合众多机构，采取联盟链的方式，引入区块链技术，从而大大降低结算成本和周期。有人预计，在全球范围，区块链应用于B2B跨境支付与结算可以使每笔交易的成本从约26美元下降到约15美元。

（3）智能合约

智能合约将是未来重要的发展方向。所谓的智能合约，就是“合约智能化”，主要特点是：合同条款不可篡改，有效性得到保障；合同制定和执行的全过程透明公开，便于监督；合同执行过程可编程，能够自动执行，不受干预。所以，这将改变未来的契约方式，促进社会更加公开公平公正。

（4）身份认证

用户身份认证与识别是银行规避反洗钱等违法行为的必要手段，各国商业银行不断投入资源加强信用审核及客户征信，耗费了大量人力物力。如果采用区块链技术，可以促进金融领域形成标准化的对用户身份数据格式的要求，从而在保证数据安全的同时实现信息共享，减少重复审核过程。据估算，在全球范围，将区块链应用于反洗钱和身份认证工作，可以为行业一年节约30亿-50亿美元的成本。

（5）电子商务

大型的电子商务网站，涉及到仓储、物流、交易和支付，每一个环节都需要大量的协作和资金结算。特别是在支付环节，目前都是基于有第三方存在的中心化的支付系统，存在着与银行资金结算慢，容易遭受各类黑客攻击等问题。使用区块链技术，通过多重签名，就可以简单的抛开第三方，实现用户与商家的直接支付，大大降低了电子商务网站的运营成本和操作风险，同时也提高了用户支付的安全性。

（6）版权保护

数字出版盗版猖獗，网络小说、网络游戏、音乐、视频、图片等资源在没有授权的情况下，被大量免费传播，与之相关的创作者和运营机构直接蒙受巨大损失，打击了优秀作者的创作激情，提高了运营机构的操作成本，阻碍了人类知识的创新发展。许多音乐艺术家、摄影艺术家、网络文学作家求助于区块链，通过区块链技术确权数字出版物，更加直接地接受读者

和用户的付费，并使用智能合约，自动解决许可问题，从而在艺术家和用户之间建立更直接的互动关系。亿书便是专注在数字出版行业，全力打造上下游生态圈，给人类创作增添新动力。

（7）证券交易

股票等证券交易结算时间较长，如果引入区块链技术，股票交易日和交割日的时间间隔可以从1-3天缩减为10分钟，交易的效率和可控性提高，交易行为的风险降低。同时，交易方身份、交易量等信息被实时记录在区块链上，有利于证券发行者提高决策效率，也有利于监管维护，减少暗箱操作。据专家预计，在全球范围，将区块链应用于现金证券的清算和结算，行业一年可以节约100多亿美元成本，而在外汇、大宗商品和场外衍生品领域，也会去除大量额外成本。

（8）贸易金融

传统的供应链金融或贸易金融业务流程高度依赖人工，包括大量审阅、验证交易单据及纸质文件的环节，不但人力成本高，各个环节出现失误的风险也很大，如果在分布式账本上管理这些流程，就可以降低人力等成本，提高效率和透明度，降低欺诈风险和人工工作失误风险。专业机构预计，在全球范围，将区块链应用于供应链或贸易金融，一年可以为金融机构和买卖方企业带来170-200亿美元的价值。

（9）物联网和大数据

物联网和大数据主要特点都是分布式存在，所以同样非常适合使用区块链技术。通过区块链技术，物联网、大数据、云存储等这类分布式系统，各个节点高度自治，彼此之间又智能协同，由于没有中央控制系统来识别对方，节点将能够独立自主地与对方进行联系，管理软件更新、软件错误或控制能源消耗，协同能力得到强化。同时安全性得到加强，不大容易引起系统性破坏和数据大范围丢失。

风险提示

如果读到这里，还对加密货币充满怀疑，您就不适合从事这行，还是赶快离开的好。如果您蠢蠢欲动，想要大干一番，那么就得把下面的内容看完，让我给你破点冷水，保持更加清晰的头脑。

加密货币仍处在莽荒的发展阶段，充满了各种神话，也存在着各种骗局，所以我说“天使与魔鬼同在”。有很多小伙伴进来，一开始会被各种诱惑所吸引，其中不乏上当受骗者，这里把本人了解的简单介绍一下，权当提醒。

（1）远离传销币

所谓传销币，上篇文章提过，就是把加密货币当作核心产品（有的甚至什么都没有，就一网站），通过传销发行的加密货币。这类币很少在主流加密货币社区出现，因为一出现就会被经验丰富的币圈大咖识破。所以，如果有人向你推销某某币，并向你承诺诱人的收益时，千万当心，你遇到的可能就是传销币，典型的如“MMM”。

这类币的诱惑性之所以非常大，是因为他们也提供真正的加密货币（从上面的分析，应该知道这个不难）。真正的加密货币都是开源的，像亿书还提供详细的源码解析，很多人都可以照猫画虎弄一个。但是，后续没有任何技术上的改进，只能是死路一条，等待崩盘或跑路。

（2）远离空壳币

所谓的空壳币，就是把加密货币当作核心产品，通过快速众筹的方式骗钱的加密货币。这个与传销币的本质相同，只不过人家公开众筹，至于会怎么样，不得而知。

这类币，突出的表现就是，从源码建立到众筹的时间比较短，各方面信息粗制滥造，一看就不是正经要干事的。每天都有很多的新币出现，有的创世贴、白皮书几乎一样，没有任何改进。这类币，最终只能以跑路为结局。

（3）避免操作风险

上面的是入行，眼睛不亮，可能就掉进坑里了。一旦找到值得信赖的产品，就会进入操作风险的层面，这里简单归纳一下：

1> 选择官方钱包

使用其他人提供的，或什么盘上下载的，可能被挂木马，损失更大。有些钱包在安装使用的时候，会被杀毒软件提示，通常需要用户手动加入白名单。这就更加充满迷惑性，被挂马的客户端基本就畅通无阻了。

2> 保护钱包私钥

私钥就像你的银行密码，不能泄漏或丢失，否则基本没办法找回。另外，加密货币的钱包私钥，比密码复杂，一大长串，很难脑记。保存在电脑上怕被盗，保存在纸上，怕丢失，但无论如何，自己要保存好，那可是自己财产的钥匙。

3> 做好钱包备份

钱包数据保存的就是自己的钱，要定期备份，以免丢失。特别是，清理电脑或更换系统时，要确保钱包数据已经备份。

上述问题看完，是不是挺受打击的？怎么这么多问题呢，这个行业还能待吗？为了回答这个问题，欢迎您移步我之前在亿书开发者群里分享过的文章《天使投资人的骗子论》(见参考)，该文的核心意思是，我的一位天使投资人朋友认为“骗子”是趋利更敏感的一群人，骗子多预示着机会也多。所以，这点小问题，只要留心，不足以成为问题。

总结

这篇文字让我重新思考了“利益”转移规则的程序化进程、趋势和风险。下一步，我们会思考，这种利益转移规则都有那些（共识机制介绍），有什么优缺点，亿书计划如何改进，请看入门部分的最后一篇:《共识机制，可编程的利益转移规则》。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

本文首发区块链俱乐部： http://chainclub.org

亿书官方网站： http://ebookchain.org

亿书开发QQ群： 185046161

参考

《请尽早把你的信誉区块链化》

《天使投资人的骗子论》

利益，网络释义

区块链到底有哪些场景和市场前景

共识机制，可编程的利益转移规则

前言

前面的文章中，我们说过，加密货币都是去中心化的，去中心化的基础就是P2P节点众多，那么如何吸引用户加入网络成为节点，有那些激励机制？同时，开发的重点是让多个节点维护一个数据库，那么如何决定哪个节点写入？何时写入？一旦写入，又怎么保证不被其他的节点更改（不可逆）？回答这些问题的答案，就是共识机制。

共识机制，可编程的利益转移规则。这个题目写出来，就有点激动，编程开发这么多年，我们尝试过很多方法，试图通过某种激励手段提高用户粘性，把用户留住。比如常见的积分机制、用户等级等，但是，没有任何一种方式，能与加密货币的共识机制相提并论。每一个区块链产品，本身就是一个小小的社会，一个由网络节点组成的自适应组织，这个组织的运行，要由共识机制来规范。

本文回答了“为什么加密货币无需监管”，主要内容包括机制的作用，加密货币共识机制的种类，它们各自的优缺点，以及亿书的改进计划。

机制，左右产品走向的根源

机制一词，原指机器的构造和动作原理，在社会学中的内涵可以表述为“协调各个部分之间关系以更好地发挥作用的具体运行方式。”

工作中遇到过一位好领导，他非常公道正派，一切事情按规矩办事，用制度说话。在他的领导下，整个部门都很有激情，心无旁骛，专心工作，上级领导也非常认可，升职加薪是常有的事情，人人都能收获满满。后来，因工作调整，我们很多人去了新部门，大家反映，再也没了当初的激情，因为工作不一定被认可，付出不一定有收获。

这个例子，可能很多人都遇到过，是最能直接体会一个社会、组织或部门当中机制运行规律的。在任何一个系统中，机制都起着基础性作用，左右着系统发展走向。在理想状态下，有了良好的机制，可以使一个社会系统接近于一个自适应系统（在外部条件发生不确定变化时，能自动地迅速作出正确反应）。正常的生物机体(如人体)就具有这种机制和能力。

机制的构建是一项复杂的系统工程。对于加密货币而言，共识机制包含各种激励制度和具体算法，比如：交易费用、区块奖励等。机制的关键因素是人，评判一个机制的好坏，往往要通过一段时间的观察，看看人参与和执行的积极性是否能够持续。如果不能持续，就预示着机制已经失败，系统或产品也将消亡。

实际上，加密货币的目标就是要建立一个“无需监管的自适应经济系统”。目前来看，支撑这个自适应经济系统的机制，常用的有三种，它们是：PoW，PoS，DPoS等，而且都能在现实生活中找到对应的经济模型。这些机制，吸引人们参与其中，组成安全网络，并有序运行。但是，长期来看，它们各有优缺点，都存在失败的可能。

下面我们看看这些机制的演进过程。

1.PoW(Proof of Work)：工作量证明机制

基本原理

这是比特币采用的共识机制，也是最早的。理解起来，很简单，就是“按劳取酬”，你付出多少劳动（工作），就会获得多少报酬（比特币等加密货币）。在网络世界里，这里的劳动就是你为网络提供的计算服务（算力x时长），提供这种服务的过程就是“挖矿”。

那么“报酬”怎么分配呢？假如是真的矿藏，显然在均匀分布的前提下，人们“挖矿”所得的比重与各自提供的算力成正比，通俗一点就是，能力越强获得越多。

优点

机制本身当然很复杂，有很多细节，比如：挖矿难度自动调整、区块奖励逐步减半等，这些因素都是基于经济学原理，能吸引和鼓励更多人参与。

理想状态，这种机制，可以吸引很多用户参与其中，特别是越先参与的获得越多，会促使加密货币的初始阶段发展迅速，节点网络迅速扩大。在Cpu挖矿的时代，比特币吸引了很多人参与“挖矿”，就是很好的证明。

通过“挖矿”的方式发行新币，把比特币分散给个人，实现了相对公平（比起那些不用挖矿，直接IPO的币要公平的多）。

缺点

一是，算力是计算机硬件（Cpu、Gpu等）提供的，要耗费电力，是对能源的直接消耗，与人类追求节能、清洁、环保的理念相悖。不过，如果非要给“加密货币”找寻“货币价值”的意义，那么这个方面，应该是最有力的证据。

二是，这种机制发展到今天，算力的提供已经不再是单纯的CPU了，而是逐步发展到GPU、FPGA，乃至ASIC矿机。用户也从个人挖矿发展到大的矿池、矿场，算力集中越来越明显。这与去中心化的方向背道而驰，渐行渐远，网络的安全逐渐受到威胁。有证据证明Ghash（一个矿池）就曾经对赌博网站实施了双花攻击（简单的说就是一笔钱花两次）。

三是，比特币区块奖励每４年将减半，当挖矿的成本高于挖矿收益时，人们挖矿的积极性降低，会有大量算力减少，比特币网络的安全性进一步堪忧。

2.PoS(Proof of Stake):股权证明机制。

基本原理

这是点点币（PPC）的创新。没有挖矿过程，在创世区块内写明了股权分配比例，之后通过转让、交易的方式（通常就是IPO），逐渐分散到用户手里，并通过“利息”的方式新增货币，实现对节点的奖励。

简单来说，就是一个根据用户持有货币的多少和时间（币龄），发放利息的一个制度。现实中最典型的例子就是股票，或者是银行存款。如果用户想获得更多的货币，那么就打开客户端，让它保持在线，就能通过获得“利息”获益，同时保证网络的安全。

优点

一是节能。不用挖矿，不需要大量耗费电力和能源。

二是更去中心化。首先说，去中心化是相对的。相对于比特币等PoW类型的加密货币，PoS机制的加密货币对计算机硬件基本上没有过高要求，人人可挖矿（获得利息），不用担心算力集中导致中心化的出现（单用户通过购买获得51%的货币量，成本更高），网络更加安全有保障。

三是避免紧缩。PoW机制的加密货币，因为用户丢失等各种原因，可能导致通货紧缩，但是PoS机制的加密货币按一定的年利率新增货币，可以有效避免紧缩出现，保持基本稳定。比特币之后，很多新币采用PoS机制，很多采用工作量证明机制的老币，也纷纷修改协议，“硬分叉”升级为PoS机制。

缺点

纯PoS机制的加密货币，只能通过IPO的方式发行，这就导致“少数人”（通常是开发者）获得大量成本极低的加密货币，在利益面前，很难保证他们不会大量抛售。因此，PoS机制的加密货币，信用基础不够牢固。为解决这个问题，很多采用PoW+PoS的双重机制，通过PoW挖矿发行加密货币，使用PoS维护网络稳定。或者采用DPoS机制，通过社区选举的方式，增强信任。

3.DPoS（Delegated Proof of Stake）：授权股权证明机制

基本原理

这是比特股（BTS）最先引入的。比特股首次提出了去中心化自治公司(DACs)的理念。比特股的目的就是用于发布DACs。这些无人控制的公司发行股份，产生利润，并将利润分配给股东。实现这一切不需要信任任何人，因为每件事都是被硬编码到软件中的。通俗点讲就是：比特股创造可以盈利的公司（股份制），股东持有这些公司的股份，公司为股东产生回报。无需挖矿。

对于PoS机制的加密货币，每个节点都可以创建区块，并按照个人的持股比例获得“利息”。DPoS是由被社区选举的可信帐户（受托人，得票数排行前101位）来创建区块。为了成为正式受托人，用户要去社区拉票，获得足够多用户的信任。用户根据自己持有的加密货币数量占总量的百分比来投票。DPoS机制类似于股份制公司，普通股民进不了董事会，要投票选举代表（受托人）代他们做决策。

这101个受托人可以理解为101个矿池，而这101个矿池彼此的权利是完全相等的。那些握着加密货币的用户可以随时通过投票更换这些代表（矿池），只要他们提供的算力不稳定，计算机宕机、或者试图利用手中的权力作恶，他们将会立刻被愤怒的选民门踢出整个系统，而后备代表可以随时顶上去。

优点

一是，能耗更低。DPoS机制将节点数量进一步减少到101个，在保证网络安全的前提下，整个网络的能耗进一步降低，网络运行成本最低。

二是，更加去中心化。目前，对于比特币而言，个人挖矿已经不现实了，比特币的算力都集中在几个大的矿池手里，每个矿池都是中心化的，就像DPoS的一个受托人，因此DPoS机制的加密货币更加去中心化。PoS机制的加密货币（比如未来币），要求用户开着客户端，事实上用户并不会天天开着电脑，因此真正的网络节点是由几个股东保持的，去中心化程度也不能与DPoS机制的加密货币相比。

三是，更快的确认速度。比如，亿书使用DPoS机制，每个块的时间为10秒，一笔交易（在得到6-10个确认后）大概1分钟，一个完整的101个块的周期大概仅仅需要16分钟。而比特币（PoW机制）产生一个区块需要10分钟，一笔交易完成（6个区块确认后）需要1个小时。点点币（PoS机制）确认一笔交易大概也需要1小时。

缺点

前几天，比特股的作者发表了一篇被广泛认为很傻的文章（见参考），预言DAO（去中心化组织）和DAC（去中心化公司）都将失败。文中披露了大量实践经验，基本算是DPoS的问题。概括起来，主要是：

一是投票的积极性并不高。绝大多数持股人（90%+）从未参与投票。这是因为投票需要时间、精力以及技能，而这恰恰是大多数投资者所缺乏的。

二是对于坏节点的处理存在诸多困难。社区选举不能及时有效的阻止一些破坏节点的出现，给网络造成安全隐患。

4.亿书对DPoS机制的改进

不过，也不可否认，DPoS机制仍是目前最安全环保、运转高效的共识机制。存在的问题，也是可以克服和解决的。针对DPoS的问题，亿书结合自己的特点，创新提出四点改进计划。

（1）熔断机制

增加反对投票功能，对于破坏节点的反对投票率达到一定数量，就会促发“熔断机制”，强制个别受托人节点降级，减少对网络的破坏可能性。

（2） 信用系统

亿书，鼓励知识分享，节点和用户之间会有频繁交互，用户对节点用户的反馈与好评，将是该节点信用积累的一部分。亿书将充分利用这些信用信息，帮助社区用户遴选优良节点。

（3） 扩大规模

101个受托人，仅仅是相对合理的经验数字。亿书，会进一步优化算法，提高网络遴选的性能，采取租赁、出售等方式，鼓励去中心化应用的开发者、出版商等第三方用户自建节点，从而更好的服务用户。

（4） 实名认证

匿名与安全是相对平衡的过程。亿书倡导提供公开、透明的服务，鼓励节点受托人实名认证，公开有关信息，接受大家监督，从而获得社区的广泛认可。对于长期表现良好的节点，亿书将给出名单列表，显示在用户帐号里。

总结

这篇文章介绍了目前大部分加密货币采用的共识机制（算法），并没有介绍瑞波币采用的瑞波共识机制，因为严格意义上来说，人们认为瑞波币不是去中心化的，同时瑞波机制除了它自己也没有发现被其他加密货币采用，所以不具备普遍性，这里就不再讨论。

实践证明，设计良好的自适应系统，必须有完善的机制作为支撑，并充分考虑人的因素。人类是复杂的，特别是人类的群体行为，更加不可预测，这些机制在最初都是设计良好，但是谁又能预料在不久的未来会出现其他的状况呢。中本聪可能也没有想到，比特币会走到今天这样。

这一篇到了设计与编码的临界点，接下来就到了不得不编码的地步了。让我们趁热打铁，通过阅读代码进一步学习研究这些奇妙的构想吧，请看下一篇：《**Nodejs**开发加密货币》之十六：**DPoS**机制的实现（源码解读）

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

首发区块链俱乐部： http://chainclub.org

亿书官方网站： http://ebookchain.org

亿书开发QQ群： 185046161

参考

为什么PoS与PoW不具有可比性

“去中心化的比特币：从自组织到专业化分工”读后感

Bitshares DPoS.

比特股创始人：The DAO 在走BTS的老路，或离死期已不远

机制解析

区块链架构设计和知识图谱

前言

区块链是加密货币背后的技术，是当下与VR等比肩的热门技术之一。最初接触区块链的小伙伴，感觉非常茫然，无从下手，原因是区块链本身不是什么新技术，类似于Ajax，说它是一种技术架构，或许更加确切。所以，这篇文章我们就从架构设计的角度，谈谈区块链的技术实现，无论你擅长什么编程语言，都能够参考这种设计去实现一款区块链产品。当然，具体到产品，架构设计有很多种，不同的人、不同的产品，架构设计也不尽相同，我们这里仅仅提供一种参考，让读者能够直观的感受区块链的技术实现，并顺便梳理与之相关的知识体系，帮助大家更进一步去学习研究。

基本概念

区块链的概念最近很火，它来自于比特币等加密货币的实现，但是目前，这项技术已经逐步运用在各个领域。什么是区块链技术？为了感性认识这个问题，我们可以使用谷歌地球的例子做类比，ajax不是什么新技术，但组合在一起就成就了产品谷歌地球，与之类似，区块链也不是什么新技术，但与加密解密技术、P2P网络等组合在一起，就诞生了比特币。技术人员，特别是Web开发工程师，学习了解ajax技术最早是被谷歌地球酷炫的效果所吸引。而现在，历史再一次重演，很多人被比特币的疯狂发展所吸引，进而开始研究其背后的技术——区块链。

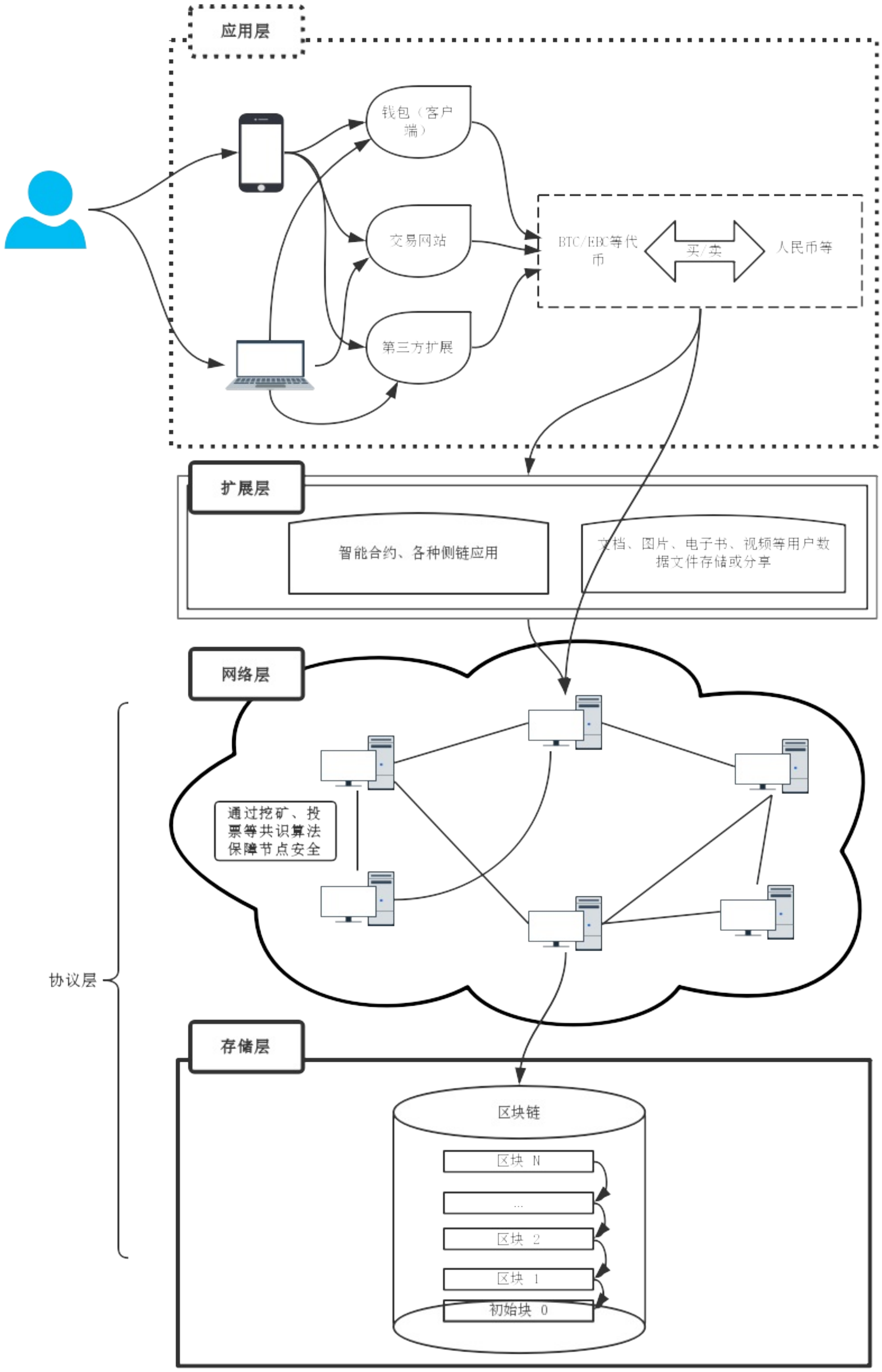
区块链作为比特币背后的技术，无需中心服务器，可实现各类存储数据公开、透明、可追溯。原本是比特币等加密货币存储数据的一种独特方式，是一种自引用的数据结构，用来存储大量交易信息，每条记录从后向前有序链接起来，具备公开透明、无法篡改、方便追溯的特点。实际上，这种特性也直接体现了整个比特币的特点，因此使用区块链来概括加密货币背后的技术实现是非常直观和恰当的。区块链是一项技术，加密货币是其开发实现的一类产品（含有代币，也有不含代币的区块链产品），不能等同或混淆。与加密货币相比，区块链这个名字抛开了代币的概念，更加形象化、技术化、去政治化，更适合作为一门技术去研究、去推广。

所以，目前当大家单独说到区块链的时候，就是指的区块链技术，是实现了数据公开、透明、可追溯的产品的架构设计方法，算作广义的区块链。而当在具体产品中谈到区块链的时候，可以指类似比特币的数据存储方式，或许是数据库设计，或许是文件形式的设计，这算作狭义的区块链。广义的区块链技术，必须包含点对点网络设计、加密技术应用、分布式算法的实现、数据存储技术的使用等4个方面，其他的可能涉及到分布式存储、机器学习、VR、物联网、大数据等。狭义的区块链仅仅涉及到数据存储技术，数据库或文件操作等。本文的区块链，指的是广义的区块链。

架构图

从架构设计上来说，区块链可以简单的分为三个层次，协议层、扩展层和应用层。其中，协议层又可以分为存储层和网络层，它们相互独立但又不可分割。如图：

区块链架构设计图



Imfly 于 2016.9.20 完成

协议层

所谓的协议层，就是指代最底层的技术。这个层次通常是一个完整的区块链产品，类似于我们电脑的操作系统，它维护着网络节点，仅提供Api供调用。通常官方会提供简单的客户端（通称为钱包），这个客户端钱包功能也很简单，只能建立地址、验证签名、转账支付、查看余额等。这个层次是一切的基础，构建了网络环境、搭建了交易通道、制定了节点奖励规则，至于你要交易什么，想干什么，它一概不过问，也过问不了。典型的例子，自然是比特币，还有各种二代币，比如莱特币等，本书介绍的亿书币也是。这个层次，是现阶段开发者聚集的地方，这说明加密货币仍在起步当中。

从用到的技术来说，协议层主要包括网络编程、分布式算法、加密签名、数据存储技术等4个方面，其中网络编程能力是大家选择编程语言的主要考虑因素，因为分布式算法基本上属于业务逻辑上的实现，什么语言都可以做到，加密签名技术是直接简单的使用（请看书中相关的加密解密文章，不建议自由发挥，没有过多的编码逻辑），数据库技术也主要在使用层面，只有点对点网络的实现和并发处理才是开发的难点，所以对于那些网络编程能力强，对并发处理简单的语言，人们就特别偏爱。也因此，Nodejs开发区块链应用，逐渐变得更加流行，Go语言也在逐渐兴起。

上面的架构设计图里，我把这个层面进一步分成了存储层和网络层。数据存储可以相对独立，选择自由度大一些，可以单独来讨论。选择的原则无非是性能和易用性。我们知道，系统的整体性能，主要取决于网络或数据存储的I/O性能，网络I/O优化空间不大，但是本地数据存储的I/O是可以优化的。比如，比特币选择的是谷歌的LevelDB，据说这个数据库读写性能很好，但是很多功能需要开发者自己实现。目前，困扰业界的一个重大问题是，加密货币交易处理量远不如现在中心化的支付系统（银行等），除了I/O，需要全方位的突破。

分布式算法、加密签名等都要在实现点对点网络的过程中加以使用，所以自然是网络层的事情，也是编码的重点和难点，《Nodejs开发加密货币》全书分享的基本上就是这部分的内容。当然，也有把点对点网络的实现单独分开的，把节点查找、数据传输和验证等逻辑独立出来，而把共识算法、加密签名、数据存储等操作放在一起组成核心层。无论怎么组合，这两个部分都是最核心、最底层的部分，都是协议层的内容。

扩展层

这个层面类似于电脑的驱动程序，是为了让区块链产品更加实用。目前有两类，一是各类交易市场，是法币兑换加密货币的重要渠道，实现简单，来钱快，成本低，但风险也大。二是针对某个方向的扩展实现，比如基于亿书侧链，可为第三方出版机构、论坛网站等内容生产商提供定制服务等。特别值得一提的就是大家听得最多的“智能合约”的概念，这是典型的扩展层面的应用开发。所谓“智能合约”就是“可编程合约”，或者叫做“合约智能化”，其中的“智能”是执行上的智能，也就是说达到某个条件，合约自动执行，比如自动转移证券、自动付款等，目前还没有比较成型的产品，但不可否认，这将是区块链技术重要的发展方向。

扩展层使用的技术就没有什么限制了，可以包括很多，上面提到的分布式存储、机器学习、VR、物联网、大数据等等，都可以使用。编程语言的选择上，可以更加自由，因为可以与协议层完全分离，编程语言也可以与协议层使用的开发语言不相同。在开发上，除了在交易时与协议层进行交互之外，其他时候尽量不要与协议层的开发混在一起。这个层面与应用层更加接近，也可以理解为B/S架构的产品中的服务端（Server）。这样不仅在架构设计上更加科学，让区块链数据更小，网络更独立，同时也可以保证扩展层开发不受约束。

从这个层面来看，区块链可以架构开发任何类型的产品，不仅仅是用在金融行业。在未来，随着底层协议的更加完善，任何需要第三方支付的产品都可以方便的使用区块链技术；任何需要确权、征信和追溯的信息，都可以借助区块链来实现。我个人觉得，这个目标应该很快就能实现。

应用层

这个层面类似于电脑中的各种软件程序，是普通人可以真正直接使用的产品，也可以理解为B/S架构的产品中的浏览器端（Browser）。这个层面的应用，目前几乎是空白。市场亟待出现这样的应用，引爆市场，形成真正的扩张之势，让区块链技术快速走进寻常百姓，服务于大众。大家使用的各类轻钱包（客户端），应该算作应用层最简单、最典型的应用。很快，亿书将基于亿书网络推出文档协作工具，这个就是典型的应用层的产品。

限于当前区块链技术的发展，亿书只能从协议层出发，把目标指向应用层，同时为第三方开发者提供扩展层的强大支持。这样做既可以避免贪多，又可以避免无法落地，是真正理性的开发路线。因为纯粹的开发协议层或扩展层，无法真正理解和验证应用层，会脱离实际，让第三方开发者很难使用。如果仅仅考虑应用层，市面上又找不到真正牢固、易用的协议层或扩展层的产品。所以，我们只好全面发力，采取完全开源开放的态度，通过社区的力量，共同去做一件有意义的事情，也算为中国区块链技术发展做点技术积累和微薄贡献。

编程实现

很多小伙伴，习惯结合自己的技术背景，来理解上面的架构设计。这里，结合具体的编程语言，简单介绍几款产品，仅供参考。

（1）C/C++

这两个语言是无法逾越的，任何开发遇到瓶颈，基本上都会找到它们，自然应该排在第一位要介绍的。同时，区块链技术的鼻祖，比特币（协议层）就是用C++语言开发的，而且目前为止，没有比比特币更加成功的区块链产品。所以，无论你使用什么语言开发，在正式进入这个行业的过程中，都应该先研究研究比特币。比特币官方客户端钱包用的Qt，第三方钱包有Python语言开发的，特别是第三方整理的开发库（Api包）很多是Nodejs设计的。比特币的架构，与上面的架构设计基本相同，另外，因为共识算法采用的是工作量证明机制（PoW:Proof

of work)，还有一些特殊的挖矿的过程。其他竞争币都是直接来自比特币的分支，所以编程语言相同，具体的技术选型和技术实现上可能有所改进，比如：莱特币，使用了其他的加密算法。

> 官方网站：https://bitcoin.org/
>
> 源码库：https://github.com/bitcoin

（2）Nodejs/Javascript

Nodejs平台强大的网络编程能力，以及js脚本语言的简单快捷，在区块链领域自然少不了它的身影。亿书便是这样一个区块链产品，亿书币是它的协议层，使用了著名的express开发框架，基于http协议开发而成。同时，它采用了授权股权证明机制（DPoS），算法上的改进，让它在处理交易时更加轻量，处理能力大大提升。它提供了强大的协作机制，为数字出版、版权保护提供了便利；扩展了侧链功能，可以基于它开发任何去中心化的应用，从而为专业作者、博客爱好者和开发者提供很多方便。《Nodejs开发加密货币》这本书完整分享了它的源码，从区块链基础概念到代码实现，从基本原理到开发设计思路，都做了比较详细的探索，目前为止，从协议层面深入代码讲解区块链技术实现的书籍极少，这算作一本。

> 官方网站：http://ebookchain.org/
>
> 源码库：https://github.com/Ebookcoin

（3）Python

如果是Python语言爱好者，我建议研究研究以太坊（Ethereum）的Python实现。尽管因为The Dao事件闹得沸沸扬扬，但从技术实现的角度来说，仍然值得参考学习。以太坊官方定位为一种开发管理分布式应用的平台，主攻方向就是“智能合约”，并为其定制了一种编程语言Solidity。以太坊的核心是以太坊虚拟机（EVM），允许用户按照自己的意愿创建操作。以太坊给出了Go、Java、Python等多语言的实现。其中以python为基础的实现主要包括三个部分：Pyethapp是客户端部分；pyethereum是核心库，实现了区块链、以太坊模拟机和挖矿等功能；pydevp2p是点对点网络库，实现了节点发现、合约代码传输、加密签名等功能，这三者组合在一起就是完整的区块链实现，后面两个核心库共同组成了协议层。另外，go-ethereum是go语言的完整实现；Ethereum(J) 是纯Java实现，它作为可以嵌入任何Java/Scala项目的库提供。客户端方面，还有Rust、Ruby、Javascript等语言的实现。

> 官方网站：https://ethereum.org/
>
> 源码库：https://github.com/ethereum/pyethapp

（4）Go

在多核时代，Go语言备受喜爱，它可以让你用同步方式轻松实现高并发，特别是在分布式系统、网络编程等领域，应用非常广。所以，在区块链开发领域，也有很多使用Go语言的项目。其中，由linux基金会主导的超级账本（HyperLeger），版本库的名字叫Fabric，就是其中一个。该项目试图为新一代的事务应用创建一种开放的分布式账本标准，支持许可式区块

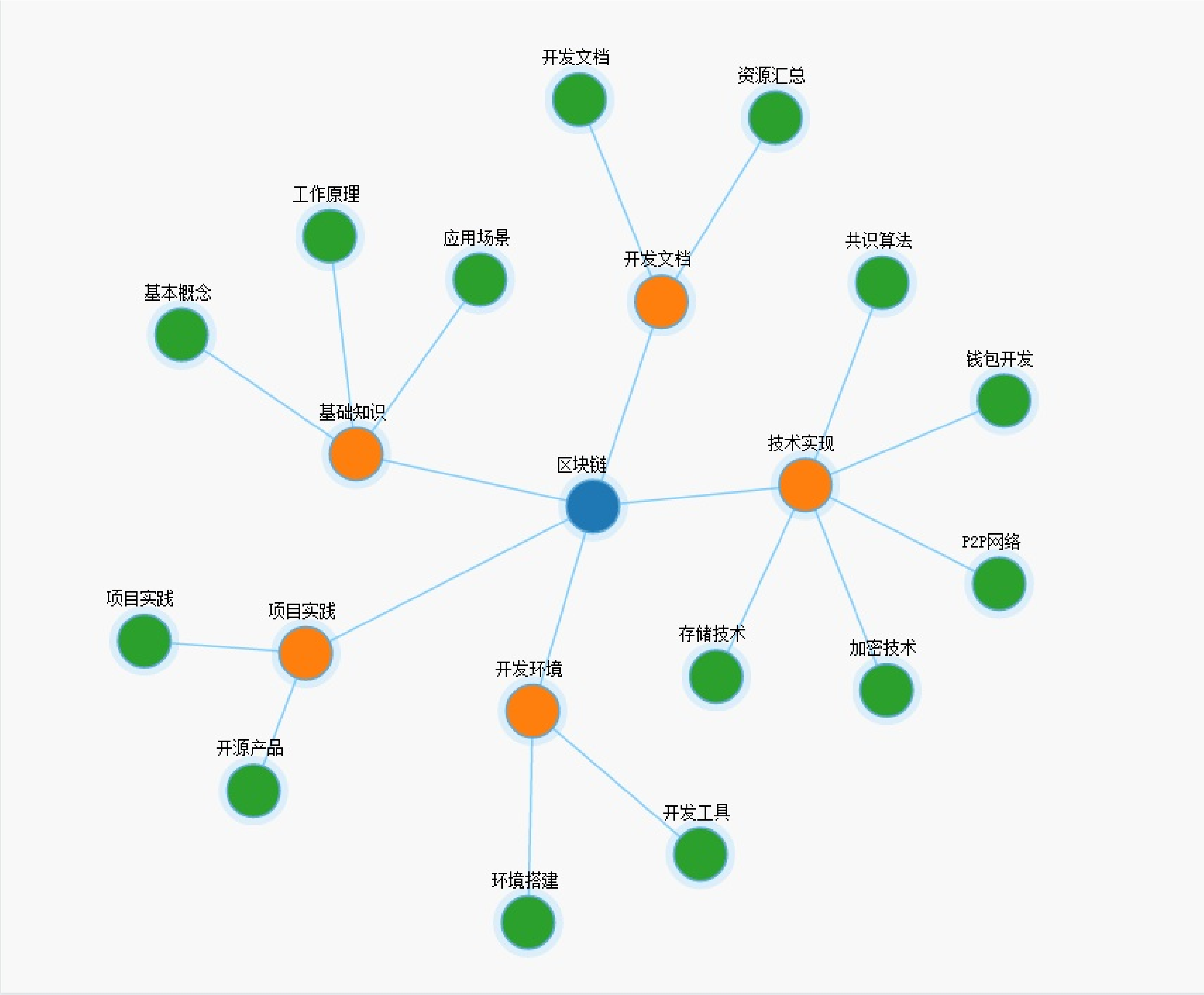
链（这种方式可能无法再现比特币那种强大的网络效应）。Fabric的开发环境建立在VirtualBox虚拟机上，部署环境可以自建网络，也可以直接部署在BlueMix上，部署方式可docker化，支持用Go和JavaScript开发智能合约。它采用PBFT分布式算法，网络编程方面用gRPC来做P2P通讯，使用 Protocol Buffer来序列化要传递的数据结构。在架构设计上，Fabric可能与比特币等区块链产品有所不同，但是上述基本组成部分还是不可或缺的。

> 官方网站：https://www.hyperledger.org/
>
> 源码库：https://github.com/hyperledger

其他编程语言，比如：C#等，也有具体实例，这里就不再列举。总之，针对不同的编程语言，在具体的编码或架构设计上可能有所差别，甚至很大，但是协议层所使用的技术并没有太大的变化。其中，网络编程是重点和难点，多数没有现成的框架可用，都是使用编程语言自身提供的库来设计开发，所以比较底层，非常考验开发者的编码功底。

知识图谱

循着上面的分析，我们已经可以了解区块链是什么，并知道怎么实现了，顺便梳理一下其中的编程技术知识，自然也就清晰多了。



根据个人的理解，我把与区块链相关的知识分为下面5个方面：

（1）基础知识

区块链是新技术，与之相关的是其背后大量的新概念、新理论。这些知识，虽然不直接体现在编码里，但却是理解区块链，掌握区块链技术的基本知识。所以，理当成为区块链技术不可或缺的一部分。这部分从基本概念入手，到工作原理的描述，就能够把区块链基础知识全部覆盖。

（2）技术实现

区块链是一项技术，但从上面的分析可以看出，它应该是一种架构应用，架构的实现理当是我们知识库的核心。正如大家看到的，任何一款区块链产品，协议层必须包括点对点网络、加密签名、数据存储、分布式算法等4个部分，应用层也必然要提供钱包、客户端浏览器等基础应用。所以，把这部分独立出来，也是合情合理。

在扩展层的部分，区块链技术可以对接各种应用，比如：金融、物联网、网络安全、版权保护、电子商务等等，现有的很多技术都可以用在这里。只不过，如何与区块链结合，如何实现跨行业使用，自然是这部分内容研究的课题。所以，这里所罗列或涉及到的技术，理应归为技术实现的一个重要部分。

（3）开发环境

区块链是多项技术的组合，有其自身的复杂性，个别应用对开发环境依赖较大，开发工具与环境搭建，是让开发者快速上手的重要内容。

（4）项目实践

据说，短短数年，全球区块链产品已经有几千个，其中不乏创新应用。有些优秀的开源产品和项目实践，是最好的学习研究资料。

（5）开发文档

这个自然不用说了，每一种产品也都会有自己的开发文档。另一个，就是有心的开发者整理汇总的一些资源，可以帮助我们节省很多查询的时间。

在考虑这个知识体系的过程中，主要思考的是，读者循着这些标签去查阅文章，能否快速掌握区块链技术，并最终上手开发实现一个区块链产品。另外，也刻意规避了与具体编程语言，以及特定领域相关的词汇，唯一可以区分的就是这些节点之下对应的文章标签。所以，这些分类就显得非常中性。也考虑过使用比特币、竞争币、智能合约、数字资产、智能资产等具体领域的实现作为分类方法，但又怕限制了读者的思维，同时随着区块链的发展，新概念将会层出不穷，那样这个图谱就需要不停的修改下去。

总结

这篇文章，我们把区块链技术基础架构描述了一下，需要再次强调的是，这仅仅是一种实现方式，绝非所有的区块链产品都是如此，我们也期待更多创新出现，也相信一定会出现。文章的编程实现部分，罗列了几种编程语言与其实现的典型产品，因为协议层技术较为底层，并没有太多现成的框架需要介绍或讨论，同时，具体的技术细节，也绝非几行字能够罗列清楚，所幸，这些产品都是开源产品，大家可以结合自己的技术背景，进一步查看对应的产品源码，很快就能了解其中的奥妙。

顺便说一下，这篇文章，是应CSDN知识库专家组邀请，为发起并筹建区块链知识库而写的推介文章，目的是帮助更多的程序员小伙伴通俗的感性的认识和了解区块链，权当抛砖引玉。CSDN作为国内著名的技术社区，始终走在技术前沿，该文最先被CSDN发布在主页头条位置，后来被巴比特论坛等多家网络媒体转载。

链接

本书源文地址： https://github.com/imfly/bitcoin-on-nodejs

官方QQ群：185046161（亿书完全开源开放，欢迎各界小伙伴参与）

CSDN发布的地址： http://geek.csdn.net/news/detail/106765

第二部分： Node.js入门指南

这部分主要针对初学者。如果您是 `Node.js` 高手，可以绕道了。

任何一门技术，都有一定的门槛， `Node.js` 也不例外。这部分，我们以解决技术选型为切入点，从开发一个统计分析工具入手，带着问题逐步深入展开。

如果您是一位初学者，其中大量的知识需要您自己去补充。因为本系列文章的目标不在 `Node.js` 本身，并且我个人的时间和水平所限，在必要的地方，我仅能作简单提示。

有什么问题，请及时反馈给我，微信：kubying

Node.js原来在币圈如此流行？

前言

本文主要讲解技术选型，币圈开源项目使用的开发语言现状，以及被程序员广泛参与的前10个有关比特币的开源项目。

开发一个产品之前，我们总会纠结要选择使用什么样的技术。考虑的因素有几个，其中包括自身所掌握的技能，项目兼容性，软硬件环境，以及应用场景等。

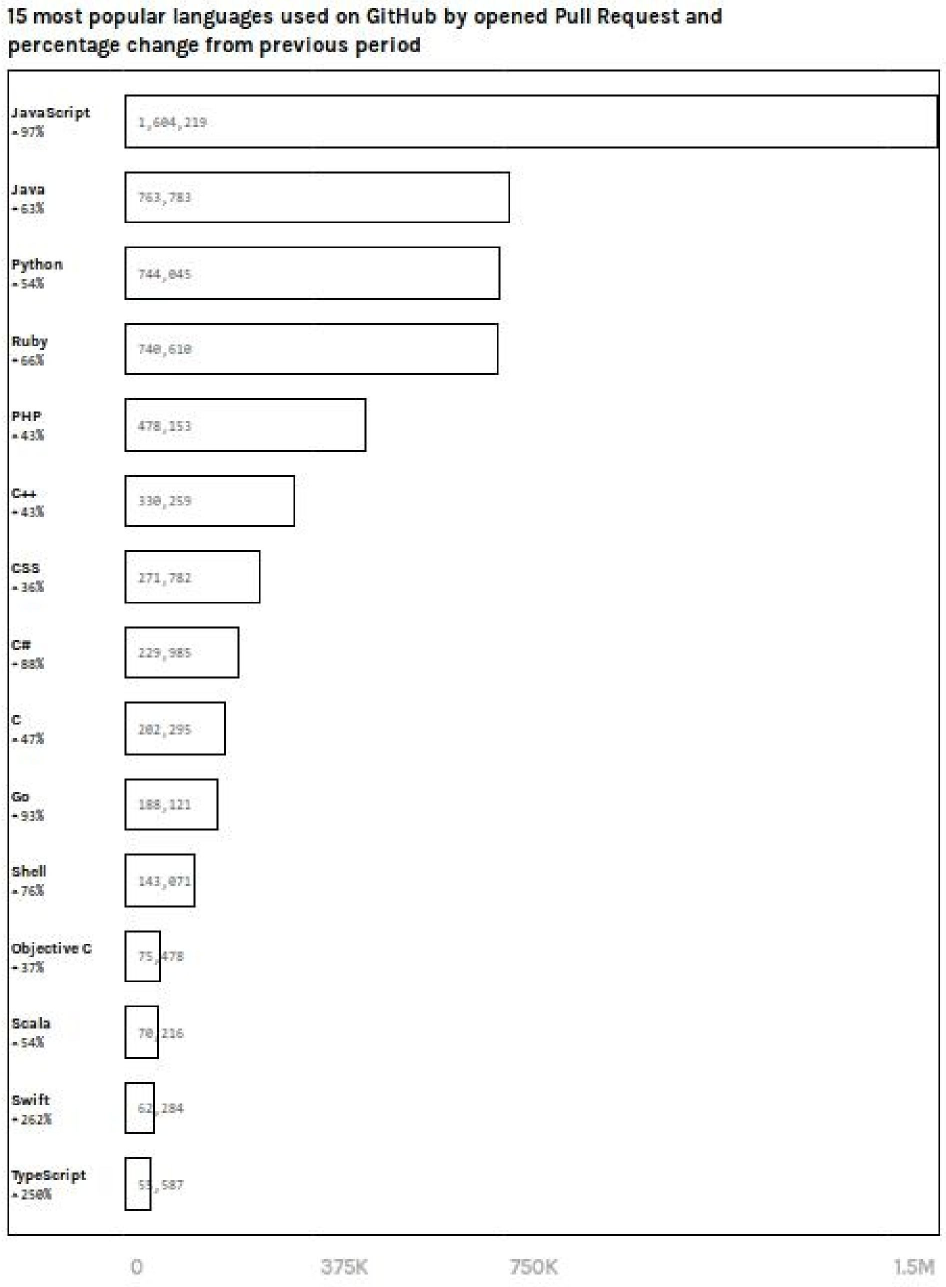
不管怎样，寻找一种通用的语言平台往往是相对合适的。这样做，可以有更多的案例学习，获得更多的社区支持，大大降低技术风险。

开发加密货币，我们计划使用 `Node.js` ，是否可行？需要尽可能的，从各个方面考察一下。

Node.js在开源社区很流行

在开源社区， Github 大名鼎鼎，我们先看看，在其上托管的项目语言使用情况。

Github上Top15项目情况（这是2016年的最新数据，见《2016年github官方报告》）



Standouts include JavaScript, C#, and Go who have seen almost doubled growth. Swift and TypeScript are up and coming with 3.5x growth.

从图中可以看出，在整个开源社区， `javascript` 被广泛应用。如果你点开其中的链接，仔细审查，会发现他们都建立在Node.js平台之上，哪怕是纯前端的项目。

这也提醒我们，目前 `javascript` 语言所对应的大部分项目都基于 `Node.js` 平台，也就是说，对于成熟的项目， `javascript` 语言大致可以代表 `Node.js` 。

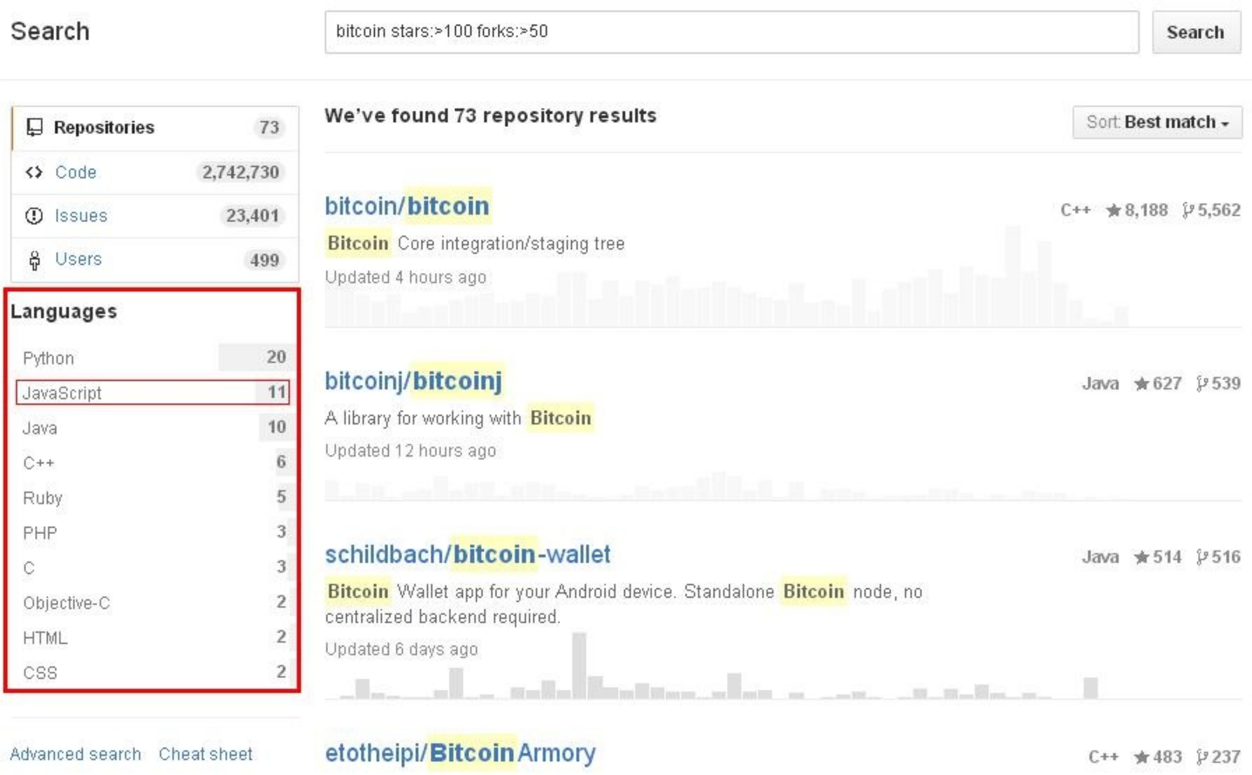
Node.js在币圈也同样流行

Github自带搜索

我们使用Github自带的搜索工具，在搜索框内输入下面的内容：

```
bitcoin stars:>100 forks:>50
```

结果如下：



使用更加复杂的查询条件，比如：

```
bitcoin OR wallet stars:>100 forks:>50 in:file extension:md
```

意思是：查询在文件扩展名为 `.md` 的文件中，包含关键字bitcoin或wallet，星100以上，fork50以上的全部项目库

其他情况，请自己试验。

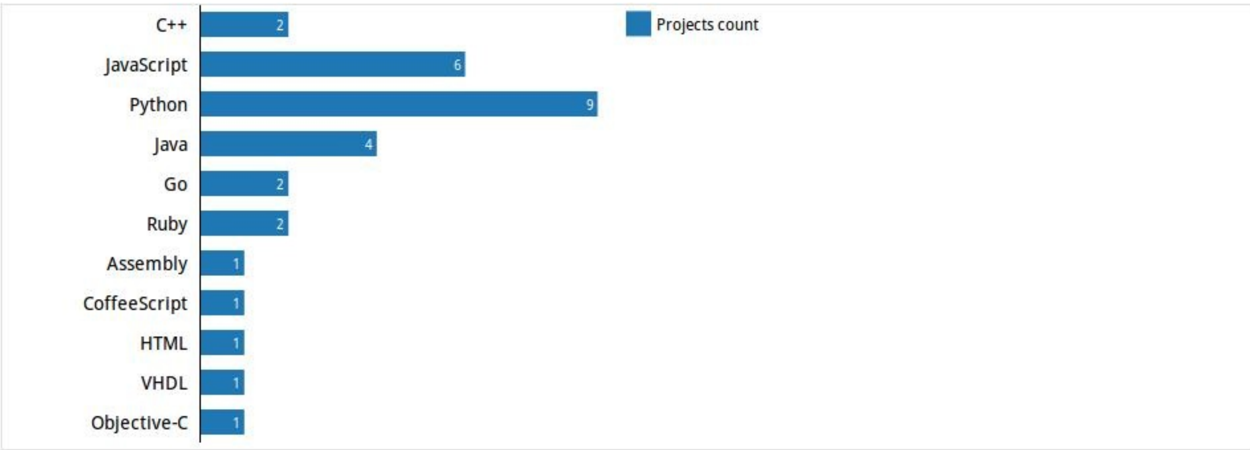
自制查询工具

程序员天生就是懒人，为了一劳永逸的获得想要的结果，我专为本文配套设计了一个小工具，看看它给我们的效果吧。

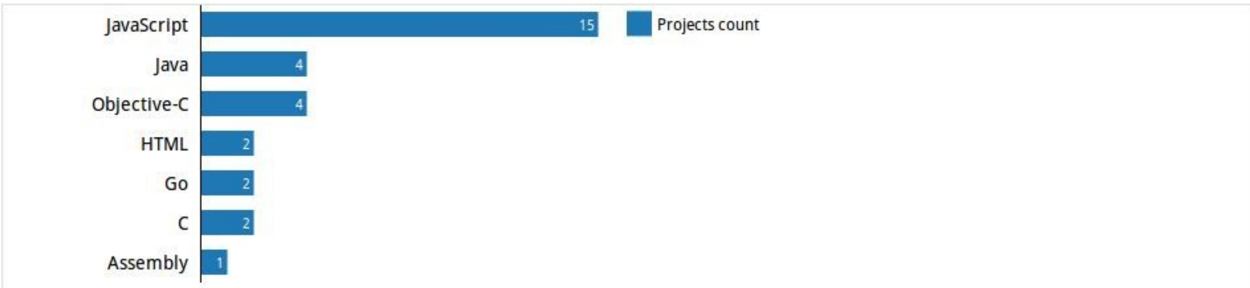
上面两个查询的对应结果如下：

(1)柱状图

查询 `bitcoin` 关键字，获得如下柱状图



查询 `wallet` 关键字（不一定是加密货币钱包），获得如下柱状图



(2)树形矩阵

查询 `bitcoin` 关键字，获得如下矩阵图



(3)更复杂的查询

请自己去体验吧，地址：https://imfly.github.io/sacdl-project。

前10个应用简介

我们使用github的搜索功能，并选择forks数量倒序排列，查询：

```
bitcoin language:JavaScript
```

注意：每一个fork背后可能就是一个全新的产品，forks代表了程序被二次开发的情况，个人觉得对于技术选型相对更有说服力。

前10个应用如下：

１. bitpay/bitcore　1656颗星，429个分支

源码网址: https://github.com/bitpay/bitcore

第一位，这是bitpay团队的产品，号称下一代PayPal。这算是一个成功案例，足见Node.js开发加密货币的可行性。巴比特有专栏介绍。

２. startup-class/bitstarter-leaderboard 295颗星，386个分支

源码网址：https://github.com/startup-class/bitstarter-leaderboard

第二位，这是一个基于比特币开发众筹网站的模板程序。巴比特在做众筹，很多人也想进入这个领域，可以参考学习。

３. bitcoinjs/bitcoinjs-lib 980颗星，305个分支

源码网址：https://github.com/bitcoinjs/bitcoinjs-lib

第三位，这是个比特币web钱包开发包，几乎当前市面上所有的基于网站的钱包都在用，牛x吧。

４. askmike/gekko 866颗星，300个分支

源码网址：https://github.com/askmike/gekko

第四位，你也想推出一个像时代、okcoin那样的基于网页的交易市场吗，这个代码不容错过。不过，我个人觉得交易市场不仅仅是技术问题，Gekko也提醒您要自担风险。

５. bitpay/insight-ui 354颗星，267个分支

源码网址：https://github.com/bitpay/insight-ui

第五位，这是bitpay放出的一个开发web钱包的UI包（要基于bitcoin-node)，看来当前开发钱包的需求还是比较大的。可以与排行第７位的bitpay/insight-api配合开发。

６. kyledrake/coinpunk　733颗星，249个分支

第六位，该项目是一个本地化的钱包服务程序，已经停止维护，取而代之的就是第３位的bitcoinjs-lib。

７. bitpay/insight-api（略）

８. cjb/GitTorrent 3065颗星，133个分支

源码网址：https://github.com/cjb/GitTorrent

第八位，不过它的好评3065颗星却是最高的。这是一个去中心化的Github，作者写了一篇博客详细解释了为什么Git也要去中心化。我本人觉得，这项目确实有意思，为我们开发去中心化的产品扩展了视野。基于这个项目思路，可以设想很多有价值的应用。

９. bitcoinjs/bitcoinjs-server

源码网址：https://github.com/bitcoinjs/bitcoinjs-server

第九位，已经放弃维护了。

10. untitled-dice/untitled-dice.github.io 26颗星，114个分支

源码网址：https://github.com/untitled-dice/untitled-dice.github.io 第十位，一个基于比特币的赌博网站源码。有意思的是，用户评价26颗星，很低，说明人们的价值观还是不喜欢赌博的。但是拷贝的分支却很多，对于开发者来说，这也算是比特币的一个落地应用。

其实，还有很多应用，没有开源，或半开源，被关注的不多，鲜为人知。

结论

仅就上述数据分析，我们可以得出如下结论:

- 在整个开源社区， `Node.js` 当之无愧是最流行的开发平台之一；
- 在钱包、交易市场等客户端应用领域， `Node.js` 的应用较为广泛；
- 在加密货币核心代码开发上， `Node.js` 的应用较少，远不如 `python` ， `java` , `c/c++` 等开发语言的使用。

不过，由于 `javascript` 有着众多的用户群，随着加密货币的发展和普及，会有更多的 `Node.js` 开发者加入。选择 `Node.js` ，就像最初选择了 `bitcoin` ，作为第一批实践的用户，我们已经站在时代潮头。

说明

本文数据收集面仍然狭窄，无法完整呈现币圈全貌，仅供参考。后续，如果发现更多更好的应用，我会持续更新，您可以关注本文的电子书形式进行跟踪。

这是一篇软文，写作不难，真正的工作量在于数据统计分析，而这也因为 `Node.js` 变的轻松加愉快。

想知道上述数据和图表怎么来吗？请看下一篇：《**Node.js**开发加密货币》之二：**Node.js**让前端开发像子弹飞一样，将简单介绍 `Node.js` 的基本使用，教您一个数据挖掘、统计分析的小技巧，并尝试去理解那些交易市场、在线钱包等实时应用的开发过程。

链接

项目源码: https://github.com/imfly/sacdl-project

试用地址：https://imfly.github.io/sacdl-project

本文源地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org/

参考

2016年github官方报告

Node.js让您的前端开发像子弹飞一样

前言

从本文开始，我们正式进入Node.js的世界。

本文，将指引您搭建Node.js开发环境，向您介绍Node.js的安装、使用，帮您快速进入Node.js的世界。

通过本文，让您对前端开发有一个完整、全新的认知，可以学习到如何将一些第三方平台的资源为己所用，比如像巴比特一样即时显示交易市场的交易行情。

本文的实例，就是上篇文章提到的加密货币开发语言统计分析项目(Statistical Analysis of Cryptocurrency Development Languages，简称 `Sacdl`)，点击这里，在线体验。

项目需求

`Sacdl` 项目需要具备以下几个功能：

- 方便地读取第三方网站（这里是github）的Api，实现项目搜索功能；
- 对读取的数据集中处理，方便地转化为我们需要的信息；
- 通过柱状图、矩阵图、表格等图表格式，将数据可视化；
- 方便扩展，为以后添加更多图表样式或其他网站Api（比如交易市场的）做好准备。

技术选型

无处不选择。大方向要选择，具体到每个开发包都要去甄别，安全吗？好用吗？性能高吗？是否有更好的方案？等等。

仅从上述需求来说，一个html文件，再加一个js文件就基本搞定，第三方包都用不着， `Node.js` 更是大才小用。

但事实上，很多仅仅是前端的项目，比如：Bootstrap等，都基于 `Node.js` ，为什么？答案很简单，它供了诸多方便实用的工具。

比如说：

- 组织方便：js没有模块化组织代码的能力。一个项目，js代码通常会分割在不同的文件中，以往的方式，处理起来非常头疼，现在利用 `Node.js` 的模块管理，可以让您彻底解脱；

- 资源广泛：`Node.js` 的出现，让js第三方包像雨后春笋一样遍地开花。需要什么，一条命令，`Node.js` 就帮您办了，这会带来极大便利；
- 全栈处理：开发完，还有很多事情要做，比如：要对前端代码js或css文件进行合并、压缩、混淆，以及项目部署等。体验过ruby on rails一键部署功能的小伙伴，都会印象深刻。`Node.js` 也很容易做到，而且更加自然、流畅。

总之，有了 `Node.js` ，我们可以像开发后台程序一样组织前端代码和项目了；有了 `Node.js` ，就有了它背后强大的技术社区支持。

Node.js简介

有小盆友说，第一次看到 `Node.js` ，还以为就是一个js文件呢。呵呵，其实，很多前端的应用，比如大家吵得最欢的 `前端开发框架三剑客` ，Angular.js， Backone.js, .Ember.js等，其实就是一个js文件。那么，

`Node.js` 是什么呢？

官方解释是这样的：

```
Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses
an event-driven, non-blocking I/O model that makes it lightweight and efficient.
```

翻译如下：

```
Node.js® 是一个搭建在Chrome V8上的JavaScript即时运行平台，采用事件驱动、非阻塞I/O模型，既轻量又高
效。
```

用句大白话解释就是，Node.js是一个可以让您利用JavaScript语言开发应用的平台, 是构建运行在分布式设备上的数据密集型实时程序的完美选择。

请注意哦，这里可没说是 `Web应用` ，很多人认为Node.js仅能开发以服务器为中心的web应用，其实不然，PC端、移动端都可以。当然，我们看到的大部分是Web应用，它是php+apache, jsp+tomcat, ruby on rails + Passenger(或thin) + nginx 等传统web开发的绝佳替代品。

如果你还没有直观感受，那么，我告诉你一个信息， `Node.js` 的作者原本是想开发一种取代apache、nginx、tomcat等产品的传统服务器软件的，结果发展成了今天 `Node.js` 的模样，你用 `Node.js` 写的每一个应用，即可以认为是一个服务器软件，也可以认为是一个web应用，而且它是如此简单、高效。

什么是数据密集型、实时应用？

聊天室、即时通信等都是。当然，所有的交易市场（比特币、股票、基金等），电子商务网站的即时交易等也是。甚至物联网，比如电器设备的监控和远程控制。本人刚完成的一个项目，是一家大型连锁超市的电器设备综合监控系统，就是使用Node.js开发的。

开发步骤

下面的过程会有点罗嗦，耐心点，很简单。

1.搭建环境

对于初学者，建议先去[Node.js官方网站][]浏览一遍。这里有币友推荐的一个中文网站，runoob.com，对于英文不太好的用户，有一定帮助。

我个人的开发环境是这样的：

- 操作系统是 `Ubuntu` 系统：您可以在现有系统上，使用虚拟机软件安装它。我们的全部示例和截图都是在ubuntu上完成;
- IDE工具： `Sublime Text` ；

（1）Node.js的安装

强烈建议参考官网信息（[Node.js官方网站][]，见参考资料）

我在Ubuntu上通过 `nvm` 安装管理 `Node.js` ，具体方法，这里有一篇详细文档，快速搭建 Node.js 开发环境以及加速 npm，请务必阅读一遍。这里摘录其中关键命令（下面的命令都要在Ubuntu的命令行程序下运行）：

安装 `Nvm`

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.29.0/install.sh | bash
```

用**Nvm**安装**Node.js**

```
$ nvm install 5.1.0
$ nvm alias default 5.1.0
```

说明：5.1.0 是Node.js版本信息，写作本文时，最新稳定版是5.4.0,长期支持版是4.2.4

安装使用**CNpm**

使用淘宝npm镜像，可以提高我们的组件下载速度

http://npm.taobao.org/

```
$ npm install -g cnpm --registry=https://registry.npm.taobao.org
```

查看版本信息

```
$ nvm -v
$ node -v
$ npm -v
```

我的版本信息如下：

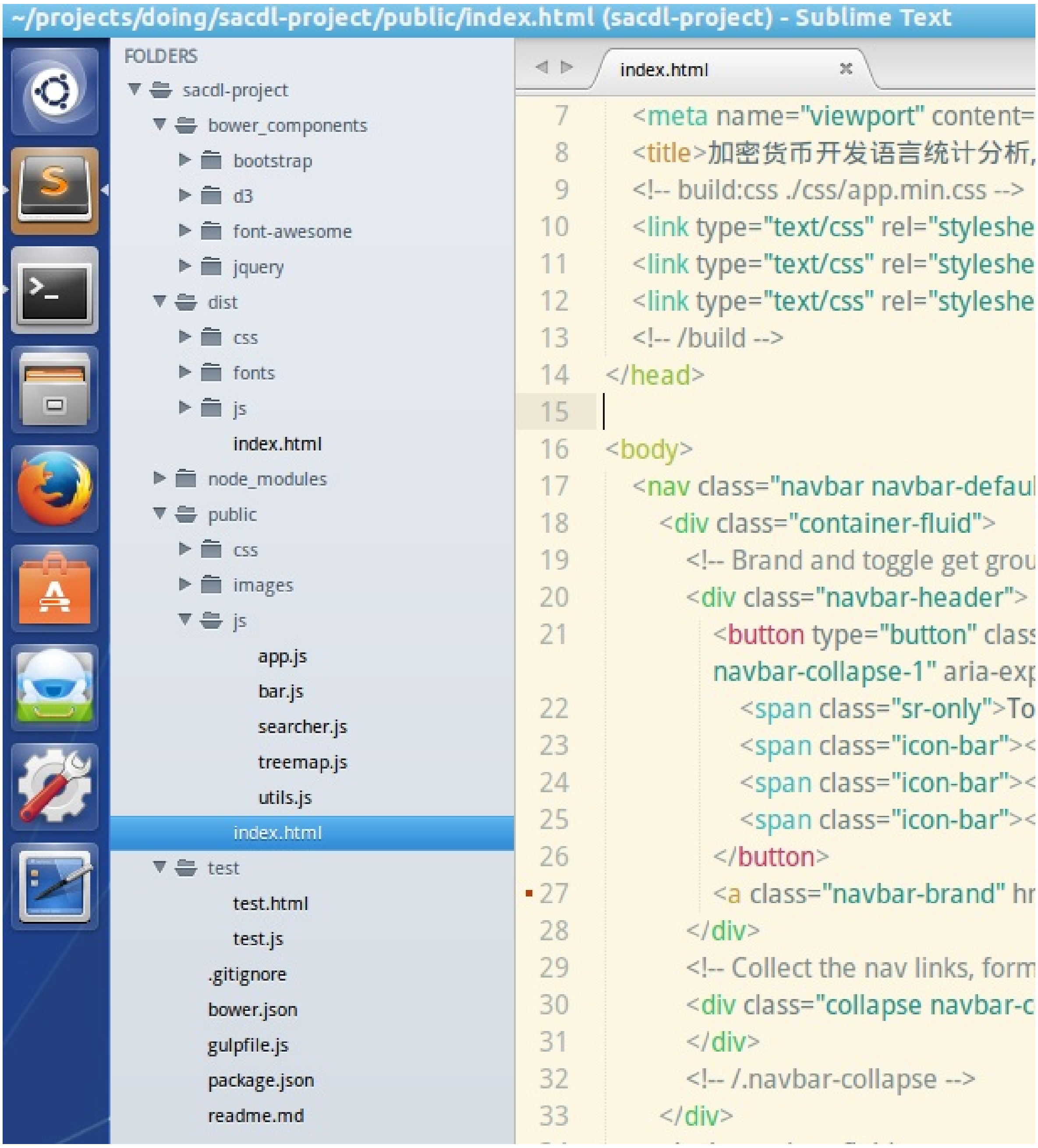
```
nvm 0.29.0
node v5.1.0
npm v3.3.12
```

2.新建工程

在您电脑上，新建一个文件夹 `sacdl-project` ，作为工程目录，路径如下：

```
/home/yourname/projects/sacdl-project
```

我们通常会把前端代码放在 `public` 目录下，然后分别建立 `js` , `css` , `images` 等目录，最后建立文件 `index.html` ， `js/app.js` ，用于显示页面和写我们的js代码，结构如下：



上述结构中，

```
`js/searcher.js`是搜索框处理代码，
`js/utils.js`是数据处理代码，
`js/bar.js`文件是d3.js的柱状图代码，
`js/treemap.js`是树状矩阵图的代码，
'js/app.js'用户综合调用，类似于控制器或路由。
```

前端第三方组件，比如d3.js等都存放在 `bower_components` ，由 `bower` 自动生成; 后台第三方模块在 `node_modules` ，由 `npm` 自动生成。

3.前端组件

在命令行，进入上述工程目录，安装前端管理工具 `bower`

```
cnpm install -g bower # 也可以使用 npm install * 命令，二者一样，只不过cnpm使用淘宝镜像，在中国安装会快些
```

说明： `bower` 是一个npm包，是专门用来管理web前端（包含js,css,images,fonts等）依赖包的。我们可以简单类比，bower用于管理前端包，npm管理后台库（包），二者用法十分相似。

初始化

```
bower init
```

结果如下：



这样会生成一个 `bower.json` 文件，这样我们的代码就被作为一个完整的前端组件来管理了。

通过 `bower` ，安装 `d3.js`

```
bower install d3 --save
```

选项 `--save` 将在 `bower.json` 文件里，写入下面的信息：

```
  "dependencies": {
    ...
    "d3": "~3.5.12",
    ...
  }
```

这样，在另一台电脑开发时，克隆完代码，就可以直接运行下面的命令，自动安装全部依赖的第三方组件了

```
bower intall
```

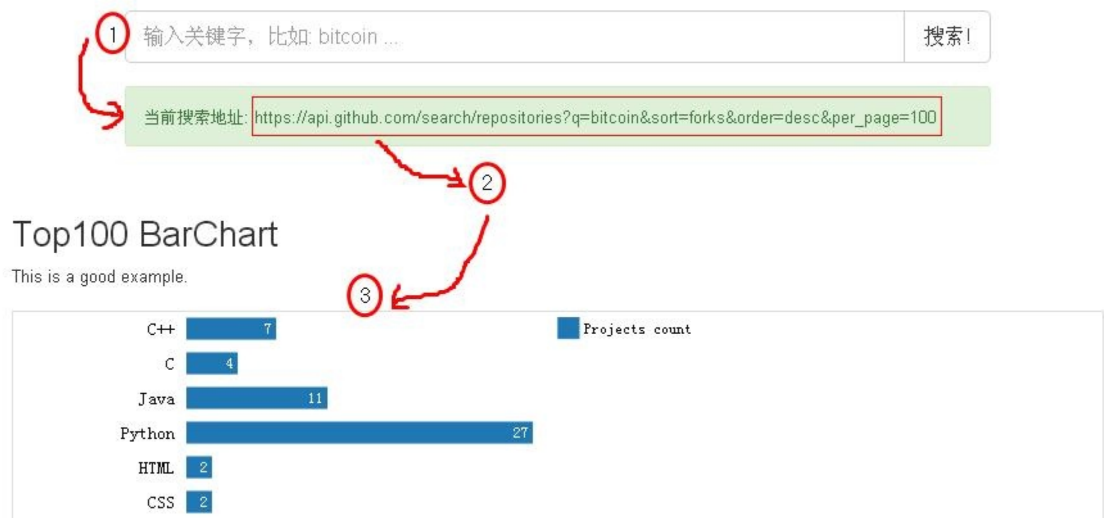
说明：`d3.js` 是提供了前端显示的柱状图、饼状图等，是数据可视化非常出名的前端开发包。国人的有百度的echarts，还有一个highcharts，这三者经常被拿来比较。简单的区分，就是，d3.js像开发包，可以任由您编程开发，但据说入门较难;其他两个更像是模板，拿来就用。

这里，我选择了 `d3.js` ，纯属个人喜好，一方面，我个人喜欢完全控制代码;另一方面，在开发电子书版权保护和交易系统，用到了它。

4.前端流程

按照上面的需求，我们的流程大致是这样的:

统计分析




```
1.接受请求：提供一个输入框，接受用户输入，获得查询关键字，并转化为github.com的Api请求地址;
2.获得数据：根据上述地址，通过ajax请求数据（这里是d3.js的d3.json()方法），对数据进行处理;
3.展示数据：使用d3.js编写图表样式，将上述数据展示出来。
```

5.学习Api

第一步非常简单，只要提供一个输入框就是了。我们直接从第二步开始研究吧。

github是用ruby on rails开发的，它的api具有典型的ror的restful风格。下面是，官方搜索示例

请求下面的地址

```
https://api.github.com/search/repositories?q=tetris+language:assembly&sort=stars&order
=desc
```

可以得到对应的json格式的数据。 官方是使用的 `curl` 命令行工具，我们直接使用浏览器即可，有图为证：


```
{
  "total_count": 340,
  "incomplete_results": false,
  "items": [
    {
      "id": 21095601,
      "name": "Tetris-Duel",
      "full_name": "Tetris-Duel-Team/Tetris-Duel",
      "owner": {
        "login": "Tetris-Duel-Team",
        "id": 7956696,
        "avatar_url": "https://avatars.githubusercontent.com/u/7956696?v=3",
        "gravatar_id": "",
        "url": "https://api.github.com/users/Tetris-Duel-Team",
        "html_url": "https://github.com/Tetris-Duel-Team",
        "followers_url": "https://api.github.com/users/Tetris-Duel-Team/followers",
        "following_url": "https://api.github.com/users/Tetris-Duel-Team/following{/other_user}",
        "gists_url": "https://api.github.com/users/Tetris-Duel-Team/gists{/gist_id}",
        "starred_url": "https://api.github.com/users/Tetris-Duel-Team/starred{/owner}{/repo}",
        "subscriptions_url": "https://api.github.com/users/Tetris-Duel-Team/subscriptions",
        "organizations_url": "https://api.github.com/users/Tetris-Duel-Team/orgs",
        "repos_url": "https://api.github.com/users/Tetris-Duel-Team/repos",
        "events_url": "https://api.github.com/users/Tetris-Duel-Team/events{/privacy}",
        "received_events_url": "https://api.github.com/users/Tetris-Duel-Team/received_events",
        "type": "Organization",
        "site_admin": false
      },
      "private": false,
      "html_url": "https://github.com/Tetris-Duel-Team/Tetris-Duel",
      "description": "Multiplayer Tetris for Raspberry Pi (in bare metal assembly)",
      "fork": false,
      "url": "https://api.github.com/repos/Tetris-Duel-Team/Tetris-Duel",
```


这就是我们得到的原始数据结构。大部分情况下，需要重新整理，不然就不用费劲开发了。这里，我们先把它转化为树形矩阵图需要的数据格式，如下：

```
{
    "name": "languages",
    "children": [{
        "name": "javascript",
        "children": [{
            "name": "imfly/myIDE",
            "watchers_count": 100,
            "forks_count": 50
        }]
    }]
}
```

这里的意思是，整个数据根节点就是 `languages` （自己建就是了）, 它以各个语言为子节点;各语言节点，则以它们的版本库为节点，这里才存储着我们需要的基本信息。

6.数据整理

我们在 `public/js` 文件夹下，新建 `utils.js` (名字随便起），然后使用文本编辑器打开，我使用的是 `Sublime text` .

（1）模块化前端代码

为了实现模块化编程，采取下面的格式组织前端代码（当然，这并不是Node.js的模块形式，不过异曲同工），

```
var Utils = (function(){
    //局部变量定义
    var a = 0;

    //公共方法
    return {
           settings: function(){},
       init: function(){},
       ...
    }

    //私有方法
    function name(){}
}())
```

在引入该文件的`index.html`文件里，就可以这样调用

```
Utils.init();
```

而无法这样调用

```
Utils.name();
```

（2）转换数据格式

如何将api读取的数据整理成我们想要的格式呢？代码如下：

```
// 一定会有一个地方传入dataset，先别着急
function getTreeData(dataSet) {
    var languages = {};

    //新建根节点
    var result = {
        "name": "languages",
        "children": []
    }

    //循环处理子节点
    if (dataSet && dataSet.items) {
        var items = dataSet.items;

        //先找出涉及到语言
        items.forEach(function(item, index) {
```

```
        if (typeof languages[item.language] === "undefined") {
            languages[item.language] = index;
        };
        })

            //根据语言进行整理
        for (var language in languages) {
        //原来有些版本库，是没有语言信息。github的语言识别并不是完美的
        if (language === "null") {
            language = "others";
        };

        //每种语言的子节点
        var root = {
            "name": language,
            "children": []
        };

        //从全局数据中再次查找我们的数据
        items.forEach(function(item, index) {
            var child = {
                "name": item.full_name,
                "watchers_count": item.watchers_count,
                "forks_count": item.forks_count
            };

            if (item.language === language || (item.language === "null" && language ==
= "others")) {
                root.children.push(child);
            };

        })

        result.children.push(root);
        }
    }

    //返回结果
    return result;
}
```

显然，这是一个私有方法。因为类似这样对数据的整理，每一个图表都要做。我们是把上面的方法作为第一步处理，然后把结果缓存，其他格式的数据都以它为基础获得（公共方法）。请参考源码 `js/utils.js` ，·点这里。

7.D3.js渲染

数据有了，终于有机会弄成我们想要的样式了。

（1）了解d3.js流程

有人说，对于初学者，d3.js的入门有点困难。如果您尝试了之后，眞觉得难，可以选择echarts，或xcharts（来自于d3.js，下面有链接），方法相同。

d3.js的基本流程是：

- 在html中，提供展示图表的位置，通常是给一个div#Id;
- 请求并填充数据;
- 渲染图表，用append()新增元素，用remove()删除多余元素;

我们用最简单的例子，演示一下（代码在工程源码的test文件夹下）：

在 `test.html` 页面添加一个div元素，如下：

```
<div id="testId"></div>
```

新建test.js文件，写下如下代码：

```
//这是要渲染的数据，可以动态获得
var dataset = [1, 2, 3, 4];

//填充数据，通常要使用d3.layout提供的数据模板进行处理，然后用data()方法去填
var chart = d3.select('#testId')
        .selectAll('p')
        .data(dataset, function(d) { return d; });

//渲染视图，主要是下面2个方法
//data（）之后才可以调用的enter()方法，意思是有数据填充的那部分图表元素，通常去增加`append`元素
chart
.enter()
.append('p')
.text(function(d, i) {
    return [d, i];
})

//data（）之后才可以调用的exit()方法，意思是无法获得数据填充的那部分图表元素，通常要删除`remove`
chart.exit().remove();
```

比如上面，我们默认提供了dataset的4个数值，第一次渲染，会正常显示4个元素;接着，数据dataset换成[5,6]，再此渲染，enter()方法将获得原来渲染[1,2]的元素，并将其值换成[5,6]，而[3,4]位置的元素因为没有了数据，被删除掉。这样就实现了图表动态转换。

注意：上面提到的 `d3.layout` 可能是一个颠覆三观的概念。 `layout` 作为层的概念，通常在html视图中用作全局共享的模板文件，比如：layout.html, layout.ejs等。但是，这里d3.js是用在数据上的，提供了 `d3.layout.treemap()` 等方法，用于对各种图表数据进行计算和处理，即： `数据模板` 。d3.js的视图处理，就是使用append()和remove()去增加或删除元素来处理，配合诸如 `.style()` 元素样式格式化的方法，实现页面控制。显然，这样做的意义就是眞正的 `数据驱动` 。

（2）渲染我们的数据

d3.js提供了d3.json(),d3.csv()等请求数据的方法，我们上述数据是json格式，自然就用前者了

我们以矩阵图为例（我也是参考官方的示例，见参考资源），在index.html加入如下元素

```
<div id="sacdlTreemap"></div>
```

然后，编写 `js/treemap.js` 代码，用于渲染图表。

最后，在 `js/app.js` 里，加载数据：

```
----部分代码------
d3.json(url, function(err, data) {
    if (err) {
    ...
        alert("加载数据失败，请检查您的网络设置。")
    };

    Utils.getData(data);

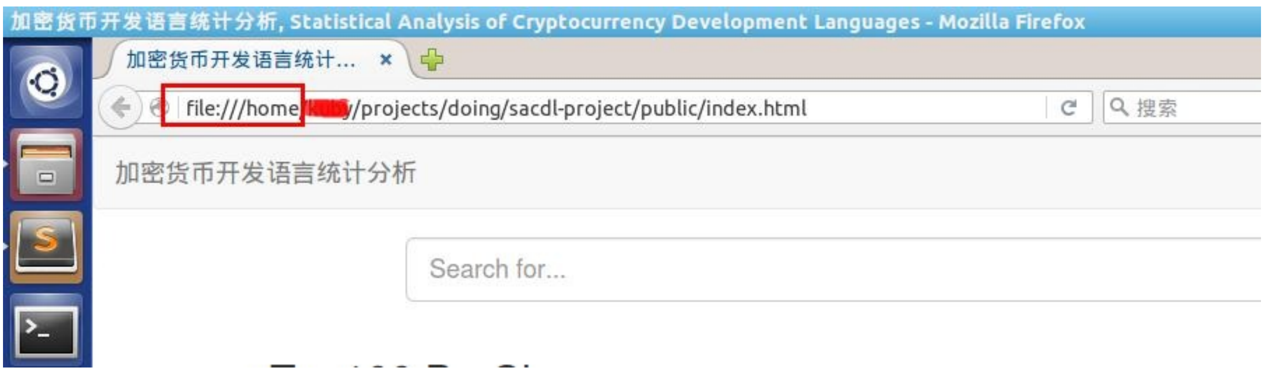
    Treemap.show();

    ...
});
----部分代码------
```

具体请看源码。

（3）查看效果

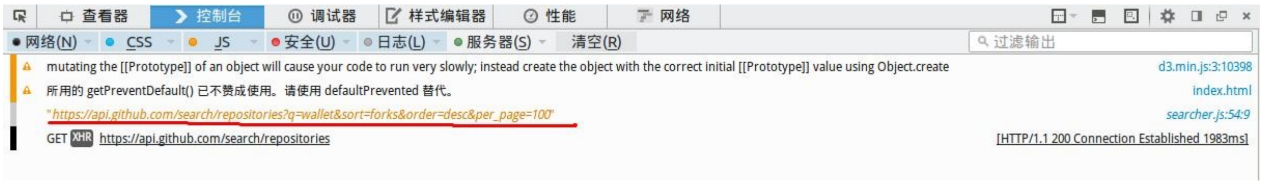
前端不用服务器，因此直接右键，选择在浏览器中打开就是了。看看效果如何。



8.代码调试

如果达到预期效果，如何发现问题所在呢？前端调试和测试也是一门学问，内容所限，无法细说。告诉您本人常用的调试前端代码的工具，就是火狐浏览器的 `firebug` 扩展插件。当然，对于本应用，用火狐或谷歌浏览器默认的控制台就可以了。

具体用法是，在打开的浏览器页面，按下 `F12` ,就会在页面底部弹出控制台窗口，如下：



错误信息，断点信息等一目了然。

9.部署发布

经过一番调试，代码终于达到预期效果。为了提高页面加载速度，增强用户体验，需要对代码进行合并、压缩，如果要保护自己的劳动，不想被别人无偿使用，还需要对代码进行混淆，最好部署到专门的服务器空间上去。这些工作，可以实现一键操作。

`Node.js` 圈子里，有2个最为流行的工具，一个是 `grunt` ,出现的最早。另一个是 `gulp` ，后来居上，号称是为了解决前者的问题而生的，目的就是为了消灭前者。事实证明， `gulp` 确实很好用，简单、高效。我们就用它。

（1）原理

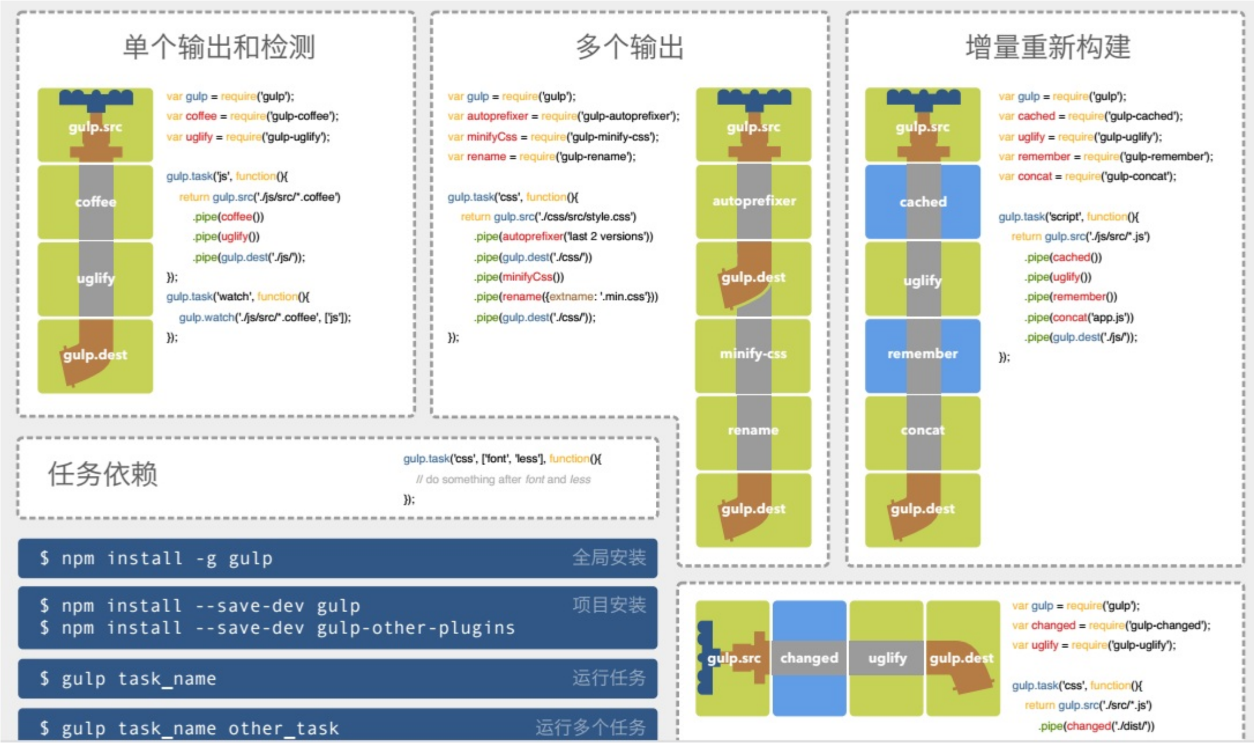
`gulp` 用到的核心概念就是管道流，你可以理解成我们生活中的各种管道的概念，比如自来水管道。文件或数据就是水， `gulp` 各类插件就是过滤网等水处理器械。

设计一个任务，就是建设一条管道，涉及到5个方法，分别是：

```
1>构建管道并起个名字用`gulp.task()`，
2>管道入口方法叫`gulp.src()`（src代表源文件），
每一节管道叫`.pipe()`(要用在入口和出口中间，在其中放入各种插件方法，就相当于加了层过滤网),
3>一直流向管道出口，方法叫`gulp.dest()`（dest英文意思是目标），
4>监控水流变化（文件变化）用`gulp.watch`,
5>综合调度各个管道的运行，用`gulp.run`
```

最后在命令行启动管道，就用 `gulp` 或 `gulp taskname` 命令

如图，看看下面的几条管道，是不是很容易理解：



注：pipe管道，是linux或 `Node.js` 等对于文件处理的一个重要概念，我们会在以后的文章中进一步说明。

（2）安装

首先，

```
cnpm install gulp --global
```

这里使用 `--global` 进行全局安装，这样我们才可以在任何路径下使用 `gulp` 命令。

然后，

```
cnpm install gulp --save-dev
```

这里安装在工程目录下，目的是方便管理。同时，因为 `gulp` 仅仅是开发辅助工具，只在本地开发机器上使用，因此上述命令添加 `--save-dev` 选项，把 `gulp` 模块安装在开发依赖里。

（3）建管道

`gulp` 命令默认请求 `gulpfile.js` 文件，手动建一个吧，上面说各类管道（任务）都在这个文件里，本工程对js进行处理的代码如下：

```
----其他代码-----
// 开建管道，名字叫`js`
gulp.task('js', ['clean'], function() {
    // 合并、压缩、混淆，并拷贝js文件
    return es.merge(                    //这是个workflow插件，是Node.js模块，都是Node.js应用
，当然也可以使用了
            gulp.src(assets.js.vendor) //管道1入口
            .pipe(gulp.dest(settings.destFolder + '/js/')), // 直接流到管道1出口，相当于简
单拷贝

            gulp.src(assets.js.paths) //管道2入口
            .pipe(order(assets.js.order)) //过滤网1：排序
            .pipe(sourcemaps.init())  //过滤网2：建sourcemaps
            .pipe(uglify())   //这算是管道中的管道了，过滤网3：混淆处理
            .pipe(concat(settings.prefix.destfile + '.js')) //过滤网4：合并处理
            .pipe(sourcemaps.write()) //建maps结束，输出sourcemaps
            .pipe(gulp.dest(settings.destFolder + '/js')) //管道2出口
        )
        .pipe(concat(settings.prefix.mergefile + '.js'))  //汇总管道：对上述2个管道的输出再
合并
        .pipe(gulp.dest(settings.destFolder + '/js/'))    //汇总管道出口
});
----其他代码-----
```

详情请看源码。

在命令行，输入如下命令，运行该任务

```
gulp js
```

（4）插件

上述代码中用到的 `order` ， `sourcemaps` , `uglify` 等对应3个 `gulp` 插件，可以从官网找到，本工程涉及到的，算是几个最常用的插件，如下：

```
"gulp-concat": "^2.6.0",         //合并js,css等
"gulp-cssnano": "^2.1.0",        //css压缩，取代了gulp-minify-css
"gulp-gh-pages": "^0.5.4",       //部署到github的`gh-pages`，本工程在线演示就是这么部署的
"gulp-imagemin": "^2.4.0",       //图片压缩
"gulp-order": "^1.1.1",          //js，css等顺序合并等
"gulp-processhtml": "^1.1.0",    //将处理完的代码，替换到.html、.ejs等模板文件里
"gulp-sourcemaps": "^1.6.0",     //产生sourcemaps文件
"gulp-uglify": "^1.5.1",         //混淆和压缩js文件
```

（5）部署

上面的插件列表里，有一个 `gulp-gh-pages` 插件，可以帮我们部署到 `gh-pages`

```
var ghPages = require('gulp-gh-pages');

//Deploy
gulp.task('deploy', function() {
    return gulp.src('./dist/**/*')
                .pipe(ghPages());
});
```

运行如下命令，即可

```
gulp deploy
```

当然，最好在运行部署命令之前，先运行合并、压缩等处理命令，如果想省事，就定义在上述部署任务里。请参考源码。

总结

写完这一章，好累。回头看看，发现上面的每一个小节，其实都可以用一章来说明。缺乏细节，会让读者，特别是新手，很辛苦。相反，太注重细节，又会让我们失去主题。所以，这也是一个很难取舍的过程，欢迎您提供宝贵意见或建议。

这里提供了完整的程序源码。源码提供的功能比文章描述的多，比如对输入框的处理、事件的监听、多数据格式的处理，还包括 `bootstrap` 的使用等。但文章仅摘录了部分核心内容，在阅读的时候，要注意结合源码，实在不明白就参考下面提供的资源，或给我留言。

现在，您应该可以自己动手试试，应该能够轻松的把比特时代、okcoin等交易市场的交易行情，即时的显示在自己的网站上了。如果，掌握些比特币核心代码，它也提供了很多Api，能不能像本文这样直接读取呢？如果可以的话，岂不是很容易就能开发一个 `blockchain.info` ？

具体分析，请看下一篇：《**Node.js**开发加密货币》之三：**Node.js**让后台开发像前端一样简单，简单介绍 `Node.js` 后台开发实践，写 `Node.js` 模块，为以后的代码分析打好基础。

链接

项目源码: https://github.com/imfly/sacdl-project

试用地址：https://imfly.github.io/sacdl-project

本文源地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org/

参考

（1）参考用例

d3.layout.treemap: http://mbostock.github.io/d3/talk/20111018/treemap.html

Grouped horizontal bar chart: http://bl.ocks.org/erikvullings/51cc5332439939f1f292

（2）官方网站

Node.js官方网站: https://node.js.org/

Bower官方网站： http://bower.io/

d3.js官方网站: https://d3js.org

（3）其他文档

xcharts一个封装d3.js的图表展示包

大数据时代的图表可视化利器——highcharts,D3和百度的echarts

d3的使用心得和学习资料汇总

Node.js让后台开发像前端一样简单

题外话

最近一直在关注比特币社区的大事件，Mike Hearn说比特币实验失败了，比特币交易价格应声大跌，币圈的朋友该如何站队，比特币的未来会如何，很多人又一次陷入迷茫。

我个人，反而更加坚定了信心。这件事充分说明，一个产品有它的生命周期，有它失败的风险，一项技术却永远前进在路上。无论产品消亡与否（当然，比特币不会那么轻易消亡），都会留下丰厚的技术遗产。

希望我的技术分享，能为这句话做个见证。

前言

上一篇文章，简单介绍了Node.js，搭建了开发环境，并轻松完成了前端开发。本文，我们更进一步，看看如何实现更加复杂的业务逻辑，如何构建自己的Api。

为什么要用后台？就我们这个统计分析项目(`Sacdl` 项目）而言，仅前台几个文件已经足够。但是，多数项目业务更加复杂，没有后台办不成事。

另外，前端处理能力有限，特别是web应用，前端代码越简单越好，对于性能和用户体验都有好处。反观我们的 `Sacdl` 项目，显然对于数据的整理更适合在后台处理。

再者，大家知道，Bitcoin或其他竞争币的核心，通常会提供Json格式的Api，我们只要在后台对这些Api进行操作，实现自己的业务逻辑，就能很轻松实现区块链浏览器(如：blockchain.info）、钱包、支付等基本应用。因此，直接学习如何处理第三方Api，对于我们快速上手基于区块链开发应用，是有直接帮助的。

需求

明确要干什么，很重要。

- 从后台读取github.com的Api;
- 处理读取的数据，并发送给前端;

很明显，我们需要重构前端代码。

开发

下文仍以 `Sacdl` 工程为例，引入Express框架，并以此为基础进行开发重构。

基于Node.js的开发框架很多，而Express是最基础、最出众的一个，很多其他的框架都是基于它构建的，比如严格模仿ruby on rails的sails框架等。

（1）安装Express

```
cnpm install express --save
```

说明：安装 Node.js 模块时，如果指定了 --save 参数，那么此模块将被添加到 package.json 文件中的 dependencies 依赖列表中。以后，就可以通过 npm install 命令自动安装依赖列表中所列出的所有模块。

（2）创建简单应用

进入工程目录，新建文件 `app.js` ，输入如下内容：

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host, port);
});
```

然后，运行下面的命令：

```
$ node app.js
```

最后，用浏览器打开 http://localhost:3000/ ，可以看到 `hello world!` 输出。

这官网的例子，是一个完整的web应用。也可以理解为一个服务器软件，不过是仅仅在3000端口，提供了一个简单的web服务。

如果，你对上篇 `gulp` 的管道概念有了一定认识，你也可以想象成，我们已经搭建了一条从后台到前端的管道。剩下的工作，就是给这条管道添加各种处理装置，让水流（数据流）实现我们的要求。关于流的概念，我们会在下一篇再次总结一下。

（3）使用模板引擎

上面，我们直接将hello world发送给了浏览器，如果是html文件该怎么做呢？Node.js没有直接渲染模板的功能，需要用到第三方插件，如:jade,ejs,hbs等

这里，咱们用 `ejs` ，它就像java的jsp，rails的rhtml，直接在html文件里嵌入代码，简单好用。下面，安装它：

```
$ cnpm install ejs --save
```

然后，用 `app.set` 设置我们的第一个管道 `过滤器` ，修改上述代码如下：

```
-------其他-------
app.set('views', './views')
app.set('view engine', 'ejs')

app.get('/', function (req, res) {
  res.render('index');
});
-------其他-------
```

我们新建views文件夹（把视图文件暂时放在这里），在views里新建index.ejs，打开它，把 `hello imfly!` 拷贝过去。

重启服务器（ctrl + C关闭，然后再用node app命令打开），刷新浏览器，看到变化，证明模板启用成功。

（4）使用静态文件服务

我们的前端，只有 `public/index.html` 文件。现在，将它的代码复制到views/index.ejs文件里，并修改js，css引用。重启，刷新，啊，一堆乱码，按下 `F12` ，打开浏览器控制台，看到一堆 `404错误` ，我们的js，css文件都没有成功加载，怎么回事？

到目前为止，我们仅提供了 `/` 地址下的路由请求，其他任何地址，Node.js都默认转向 `404错误` 。怎么办？一个个添加吗，显然不是，这类静态文件，express提供了简单的方法：

```
app.use(express.static('./public', {
    maxAge: '0', //no cache
    etag: true
}));

app.get('/', function (req, res) {
...
```

这是咱们使用的第二个管道 `过滤器` ，上面的代码意思是，在public下的文件，包括js,css,images，fonts等都当作静态文件处理，根路径是 `./public` ,请求地址就相对于 `/` ，比如：./public/js/app.js文件，请求地址就是http://localhost:3000/js/app.js

说明：这里有一个小问题，使用bower安装的前端第三方开发包，都在bower_components文件夹下，需要移到public文件夹里。同时需要添加一个.bowerrc文件，告诉bower组件安装目录改变了，并修改gulpfile.js文件。当然也可以连同bower.json文件都拷贝到public文件夹里。

重启服务，刷新页面，我们看到了久违的页面。



（5）后台请求githubApi

在后台请求github也有很多方案，使用Node.js的request插件，就可以直接在后端请求http地址，相当于直接把前端代码拿到了后台。不过，这里有个更好的方案，github提供了Node.js使用的开发包，我们可以直接用：

```
cnpm install github --save
```

官方地址：https://github.com/mikedeboer/node-github

为什么会想到这个方案？一个方法是，认真阅读官方文档，看它提供了什么资源;另一方法是，京城去 `https://npmjs.com` 上搜搜。通常，成熟的产品，一般都会提供现成的方案。

该组件集成了githubApi的几乎全部内容，当然包括搜索功能。下面，让我们试试，把下面的代码拷贝到 `app.js` ：

```
var GitHubApi = require("github");

//下面的代码放在app.get('/', ...)之前
app.get('/search', function(req, res){
        var msg = {
        q: 'bitcoin',
        sort: 'forks',
        order: 'desc',
        per_page: 100
    }

    github.search.repos(msg, function(err, data) { //这里必须用`回调`函数，不能使用 var dat
a = ...的方式，下篇细说
        res.json(data); //输出json格式的数据
    })
})
```

在浏览器里请求 `http://localhost:3000/search`，可以看到与请求 `https://api.github.com/search/repositories?q=bitcoin&sort=forks&order=desc&per_page=100` 一样的结果（样式可能不同）

`http://localhost:3000/search` 就是我们自己的Api服务，如果有自己的数据库或其他逻辑业务，数据很容易融合进去。

让前端请求这个api，修改public/js/app.js 34行的代码，如下：

```
url = url || 'http://localhost:3000/search'; //默认请求的页面
```

在浏览器里请求 `http://localhost:3000/` ，结果与原来相同。这样我们就简单实现了，后台处理数据，前台展示数据。

（6）模块化重构

到目前为止，我们都是在修改 `app.js` ，不断往里面添加各种管道 `过滤器` 。如果业务复杂，这个文件会非常大。事实上，任何一个Node.js应用，都可以压缩成这样一个js文件。这就能轻松理解，为什么多数js框架都习惯带着js后缀了吧，因为它本身就是一个js文件。

但，这样不适合开发和维护。我们需要把它分解成一个个独立的文件，通过名字就能分辨它的用处，通过名字就能直接用它，这就是Node.js的模块化开发。

Node.js的模块化非常简单。记住这样一个简单的逻辑关系：通常一个module.exports可以定义一个模块;一个文件只包含一个模块;只要是模块就可以使用require()方法在其他地方引用。样式与我们的前端代码非常相似，如下：

```
//文件 /path/to/moduleName.js

//局部变量
var a = ''

//公共方法（直接导出模块）
var moduleName = {} 或 function(){} //总之就是一个对象

//私有方法
function fun1(){}

//导出模块
module.exports = moduleName;
```

在其他文件中，我们可以这样用：

```
var moduleName = require('/path/to/moduleName'); //默认js后缀，习惯不用带

//然后，直接调用moduleName的各对象或方法就是了
```

按这个方式，我们把app.js文件，拆分成典型的 `MVC` 的开发样式，比如，熟悉ruby on rails的朋友，都习惯把代码分开保存在controllers,models和views文件夹里，以及router文件，这里，我们仿效之。视图已经定义在了views文件夹下，下面我们重点拆分其他的。

a）拆分模型

模型model专门处理数据，无论是数据库，还是请求远程api资源，都应该是它的事。自然，我们可以把githubApi的请求独立出来，这么做：

新建文件夹和文件 `app/models/repo.js` ,剪切粘贴下面的代码

```
var GitHubApi = require("github");

/**
 * from https://www.npmjs.com/package/github
 * @type {GitHubApi}
 */
var github = new GitHubApi({
    // required
    version: "3.0.0",
    // optional
    debug: false,
    protocol: "https",
    host: "api.github.com", // should be api.github.com for GitHub
    pathPrefix: "", // for some GHEs; none for GitHub
    timeout: 5000,
    headers: {
        "user-agent": "My-Cool-GitHub-App" // GitHub is happy with a unique user agent

    }
});

var Repo = {
    search: function(msg, callback) {
        var msg = msg || {
            q: 'bitcoin',
            sort: 'forks',
            order: 'desc',
            per_page: 100
        }

        github.search.repos(msg, callback);
    }
}

module.exports = Repo;
```

说明：模型Model是资源的集合，就像数据库里的一张表，名字自然用资源类的名词表示最好。对数据的增删改查都在模型里，自然在前端 `public/js/utils.js` 里的部分代码就应该转移到这里，从而直接输出treemap的数据格式，例如：

```
// from /public/js/utils.js
function treeData(data) {
    var languages = {};

    var result = {
        "name": "languages",
        "children": []
    }

...
```

b）拆分控制器

控制器负责从模型请求数据，并把数据发送到前端，是前端和后台的`调度员`。这里，`app.get`方法里的匿名函数便是，我们分别把他们抽取出来，放在`app/controllers/repos.js`里，并把请求githubApi的代码用模型代替，代码如下：

```
var Repo = require('../models/repo');

var Repos = {
    //get '/'
    index: function(req, res) {
        res.render('index');
    },

    //get '/search'
    search: function(req, res) {
        Repo.search(req.query.query, function(err, data) {
            res.json(data);
        })
    }
}

module.exports = Repos;
```

说明：按照惯例，控制器的名称，通常是对应模型的名称（名词）的复数;行为的名称，通常是动作（动词），因此`repos.index`就表示版本库列表，`repos.search`就是搜索版本库信息，在`app.js`中，自然这样调用：

```
----其他代码----
var repos = require('./app/controllers/repos');

app.get('/search',  repos.search);
app.get('/',  repos.index);

----其他代码----
```

这部分代码，起到分发路由的作用，一看便知应该在路由里，于是继续重构。

c）拆分路由

新建文件 `app/router.js` , 把上面的代码剪切过来，修改为：

```
var repos = require('./controllers/repos');

var Router = function(app) {
    app.get('/search', repos.search);
    app.get('/', repos.index);
}

module.exports = Router;
```

在 `app.js` 里，简单调用:

```
var router = require('./app/router');

router(app);
```

以后，再添加其他的任何路由，只要修改router.js就是了。

d）整理视图

把views整体移动到 `app/views` 下，并修改 `app.js` 代码，让模板引擎指向该文件夹

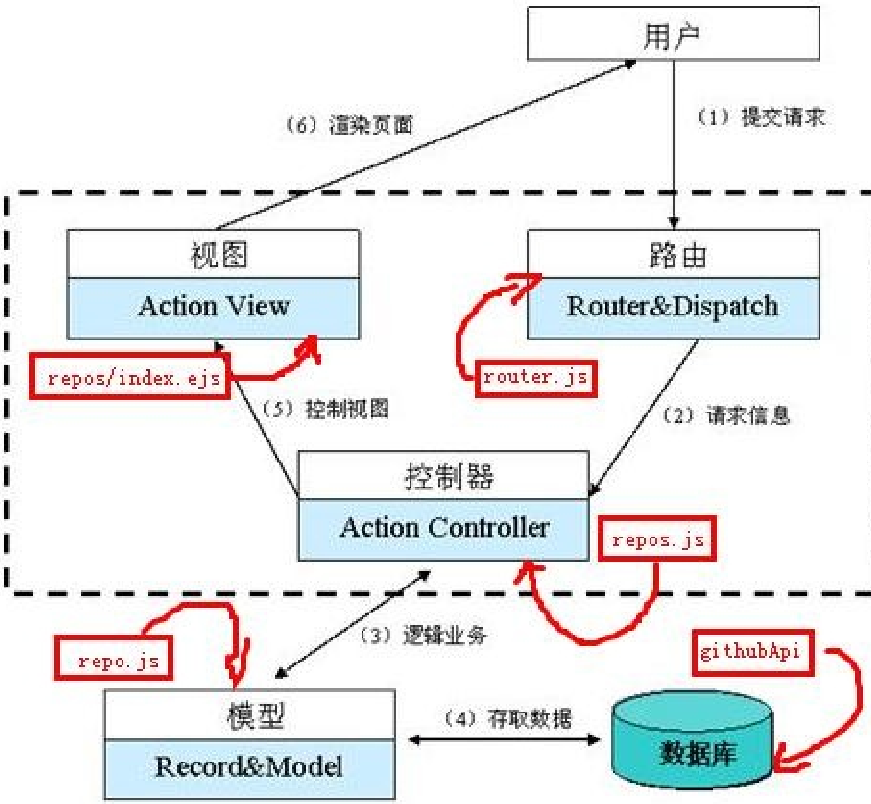
```
app.set('views', './app/views')
```

然后，新建 `app/views/repos` 文件夹，将 `views/index.ejs` 文件移入。

说明：视图view是界面元素，通常是类html文件，按照惯例，它的文件夹与控制器同名repos，各文件名与控制器的行为action同名，如index -> index.ejs

当然，视图根据模板引擎的特点，也可以进行模块化处理，进一步细化为layout.ejs, header.ejs等，方便重复使用。限于篇幅，不再罗嗦。详情，请看源码。

经过这样的模块化整理，我们轻松实现了一个简单的 `MVC` 框架（如图），它的易用性、扩展性得到很大提升。我们已经可以快速添加更多的功能了，比如：像巴比特那样显示主流交易市场信息（下篇）。



（7）测试（略）

无测试不产品。不过咱们还是省了吧，不然，又要很长的篇幅。本项目仅作文章辅助，不能作为产品使用。如果非要用，建议您写写测试，不然后果自负。

说到这的时候，本人就有一个问题没有解决：我的办公室有代理，通过Node.js请求api.github.com是不成功的，但在家里就可以。这个问题，就留给高手去解决吧。

我们的方向是文章，每个细节都想完美，最终的结果就会不完美。很多程序员在某个时期，会不自觉的陷入技术细节，而忽略很多重要的东西。最严重的是，很多人始终都没有拿出过一个完整的产品，像样的更不用说了。“使劲看就是盲，太专注就是愚”，值得深思。

（8）部署（略）

自己找个服务器，折腾折腾吧。

总结

本文涉及的代码非常简单，但在严格控制字数的情况下，仍然罗嗦了这么多。所以，很多时候，语言的力量非常苍白，说这么多干嘛，做就是了。

文中，对输入框的处理没有说明，请自己查看源码（写到这个时候，其实还没有做）。模块化部分，其实已经有一个叫express-generator的插件，可以一键生成，所以很多优秀的资源，自己去探索吧。不过窃以为，该说的重点基本提到。

总体来说，使用Node.js，从前端到后台并不复杂，或者说非常简单。前后语言的统一，给我们减少了很多思维转换的麻烦。如果你能完整实践这两篇文章，我个人认为，Node.js应该可以入门了。

但是，要想看明白一些复杂的代码，还需要掌握Node.js的一些独特习惯，比如回调，比如对异常的处理等等，请看下一篇：《**Node.js**开发加密货币》之四：您必须知道的几个**Node.js**编码习惯，仍以sacdl项目为例，添加展示主流交易市场信息等功能，目标是简单介绍一些大家经常遇到的坑，以便在接下来的代码分析中更加轻松。

链接

项目源码: https://github.com/imfly/sacdl-project

试用地址：https://imfly.github.io/sacdl-project （前端）

本文源地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org/

参考

Expressjs官网： http://expressjs.com/

您必须知道的几个Node.js编码习惯

前言

前面的两篇，以 `sacdl` 工程为例，简单介绍了Node.js的环境搭建和代码组织。这一篇，做个简单的小结，把涉及到的编码习惯用我个人的理解，提示性的说明一下。

编程，其实就是用 `特定的语言` 讲故事、写规则。 `特定` 就是习惯，就像中国的方言，掌握了技巧，很快可以交流，剩下的细节慢慢积累就是。

比特币体现了人类去中心化的本质，Node.js也是最能体现人类特质的编程语言之一，比如：一切都是数据流，事事皆回调。下面，让咱们慢慢品鉴一下吧。

1、一切都是数据流

概念理解

`Node.js` 默认提供了 `Stream` 模块，即：流，我们在第二篇讲解gulp的时候说起过。作为一个抽象接口，它是Node.js的基础模块，被其他很多模块所使用。

流，最早是linux环境下的概念，与之对应的方法经常是pipe，即：管道（方法）。我个人理解，这种设计非常形象合理。流的概念存在于人类生活的任何场景，数据流，与水流、空气流等具有相同的特性。如：

（1）流，兼具时间和地点两个坐标，时间代表一个过程，地点代表流入、流出（对应发生、结束的编码）位置;

（2）流，是一个时间上的线性过程，而非一个时间点。在时间段内的传输只是部分数据，若要获得完整的数据，就需要花费足够的时间;

（3）流，只能沿着构建在从发生到结束的编码位置的管道传输，在传输的过程中，可以被调整和改变;

再直白一些，流，不可能一下子发生或结束，再快也得有个时间差。就像人类社会，始终以时间为单位，这一刻到下一刻，已经发生变化。而Node.js严格尊重这个现实，无论是远程访问，还是本地请求，每一个data都被分成一段一段数据流（通常是Buffer对象）传输。

因此，Node.js里没有简单拷贝的概念，或者说拷贝其实可以通过流来简单实现。

代码示例

比如：

```
//引用模块
var fs = require('fs');

var rs = fs.createReadStream('test.md');
var chunks = [],
    size = 0;

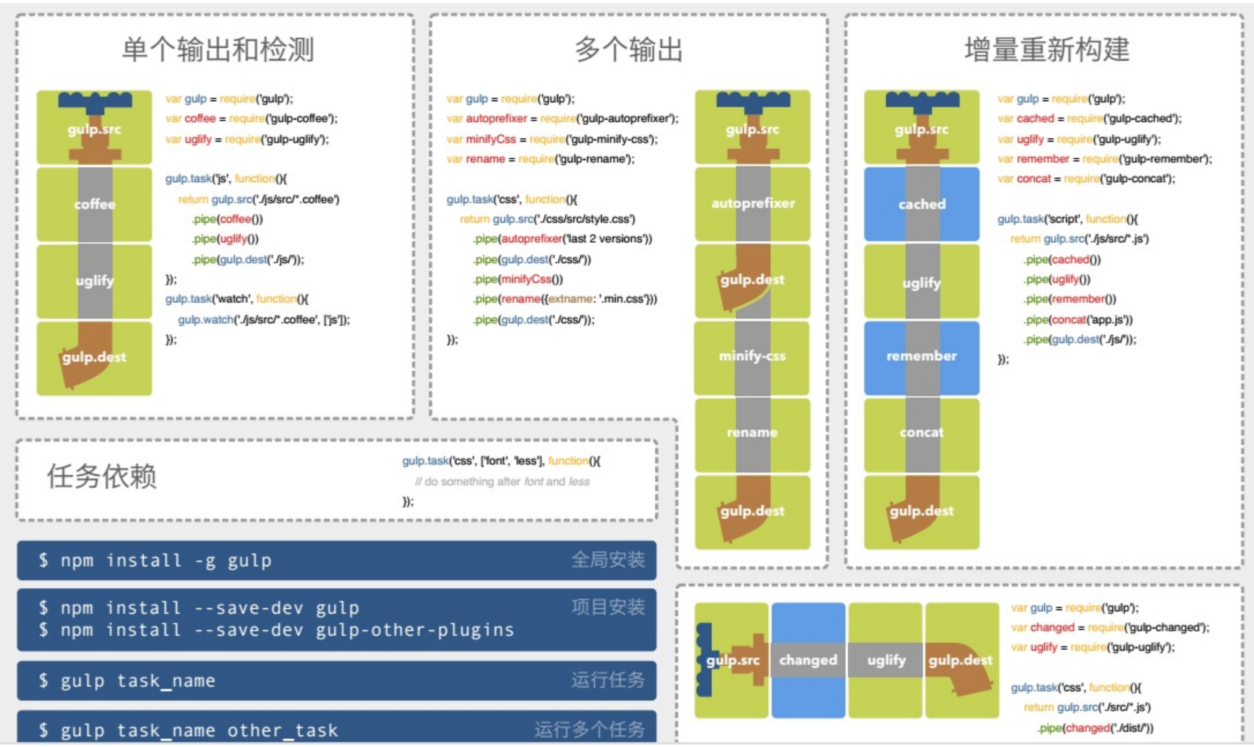
//接受数据：一段段的接受
rs.on("data", function (chunk){
  chunks.push(chunk);
  size += chunk.length;
});

//拼接数据，并转换为字符串（**注**：如果 test.md 不是 UTF-8 编码格式的，如果直接
//转换为字符串，可能会出现乱码，这时需要使用其他的模块（比如：iconv-lite）来进行转换）
rs.on("end", function() {
  var data = Buffer.concat(chunks, size);
  var str = data.toString("utf8");
  console.log(str);
});
```

其实，在Express的编码中，默认也是在处理buffer数据流，上面的方法照样适用。

思维习惯

记住，在Node.js的世界里，无论是处理文件，还是请求远程资源，处理的就是“数据流”，处理方法都是如此。为了直观，还是用第二篇里那张 `管道` 图吧：



进一步：因为任何 `流` 都是时间的函数，为了节省时间、提高效率，最好的方式，当然是 `并行` 处理。也就是说每一个 `流` 最好使用一个独立的线程，而不影响其他的。事实上，Node.js就是这么处理的，这就是下面要说的 `回调` 。看看上面的代码形式，就是典型的 `回调` 用法。

2、事事皆回调

概念理解

中文 `回调` 这个词，总不能让人一下子就理解得那么透亮。其实，如果按英文 `callback` 直译最好， `调回` 虽然直白，但好理解。

“事事皆回调”，是不是有点绝对？其实不然，它贯穿在人类沟通交往的全过程。大到工程项目，小到求助问路，只要是你必须或需要别人帮你干的事，就得用到 `回调` ，即：让他人完成并获得明确的回应。让别人为你做某件事，叫事件触发;给一个方法获得该事结束后的结果，就是 `回调` 。

举个典型的示例：假如你是一位老板，很多事情需要下属去做，然后获得结果。如果下属合格，安排的事情，他（她）会按照要求给你回应，即便没有达到预期的结果，聪明的下属也会及时向你汇报进展。但现实普遍存在的现象是，老板安排的很多事情并不能得到及时回应，这时候你就得亲自过问，要不打电话、要么派秘书或办公室人员去追踪，总之，要有一条 `通信线路` 让你得到回馈（无论好坏）。

这里，优秀员工主动汇报工作的行为，以及老板打电话或派人的行为，都是 `回调` 方法。很容易想象，假如没有这些 `回调` ，我们还能做成什么事，世界会是什么样。一些国字号的企业，为什么效率低下？几个人干着其他公司一个人的活？社会批评说是懒散，是不干活吗？不尽然，其中大部分是不给你 `回调` （或者给你不明确的 `回调` ），让你无从干起。

因此，一个优秀的老板，一定是一个会选人、用人和培养人的人，如果员工只知道埋头干活，不知道及时反馈信息，老板一定会有自己的招式应对;一个优秀的员工，一定是一个让老板节省脑力的人，事前、事中、事后都会适时汇报情况。总之，只要想做事、做成事，一个企业或组织会自觉地对信息 `回调` 的方法进行优化，降低沟通成本，提高效率。

具体到Node.js的程序，我们形象的比喻一下就是，主线程是老板，子线程（ `线程池` ）是员工，而 `回调` 就是它们之间的通信方法。使用 `回调` 的代码，我们称为 `异步` 编程。说白了，就是你干自己擅长的、份内的事，其他的都交给别人去做。即：对主线程而言，是 `异步` ，而从局外人来看，是在 `同步` 做事情。

注意：Node.js是单进程的（而非单线程，官网解释的很清楚），只不过我们写的js代码仅在单（主）线程运行罢了， `回调` 都放在事件轮循里处理，而事件轮循等底层代码跑在多线程上（即大家公认的线程池）。这部分讨论，我们会在后续章节，比如加密解密部分，结合如何编写CPU密集型程序去深入讨论。

代码示例

具体形式，在前面的文章中用过多次了，这里再举个例子，类比一下：

（1）在前端开发时，我们用d3.js通过Ajax的方式请求数据的代码，可以简化为：

```
d3.json('/resource.json', function(err, data){
   //code
   console.log("Hello, ", data);
});

console.log("I`m end.");
```

这里，我们请求的 `/resource.json` ，其实就是远程服务器上的资源 `http://localohost:3000/resource.json` ，后面紧跟的函数就是 `回调` 函数，要在资源请求结束之后，才会调用。所以，`Hello，... `必然要出现在` I'm end.`之后。

(2)在后台开发时，我们也有与之几乎一样的代码：

```
var fs = requrie('fs');
fs.readFile('/resource.json', function(err, data){
   //code
   console.log("Hello, ", data);
});

console.log("I`m end.");
```

这里，我们请求的 `/resource.json` ，其实就是本地的资源 `resource.json` 文件，后面紧跟的函数也是 `回调` 函数，同样要在资源请求结束之后，才会调用。所以，`Hello，... `也会出现在` I'm end.`之后。

思维习惯

在Node.js的世界，到处是 `回调` （多数使用callback、next、cb等命名回调函数），到处是 `异步` ，当你不自觉的编写了下面的代码，而反复调试，得不到 `data` 预期的结果时，要意识到，您已掉进了 `异步` 的陷阱，忘记了 `回调` 。

```
var fs = requrie('fs');
var data = fs.readFile('/resource.json'); //异步方法

//code
console.log("Hello, ", data);

console.log("I`m end.");
```

当然，可以这样修改上面的错误代码：

```
var data = fs.readFileSync('/resource.json'); //同步方法
```

在很多其他编程语言里，就是这么用的。这样做的好处，就是直观，便于人类直线思考。坏处就是，数据（流）大时，必然需要长时间执行，直接`阻塞`进程，整个程序只好停下来等着，这就是`I/O阻塞`。

Node.js因为用了`回调`，js代码所在的（主）线程会把一切`回调`扔给后台的线程池去处理，而自己一步到底，所以叫`I/O非阻塞`。

进一步：既然“事事皆回调”，那么`回调`里面也可能有`回调`，事实上，这种情况非常多。这也是很多人对Node.js畏惧的原因之一，多年的高手也经常栽在上面，于是大家总结有`回调大坑`的说法，就是回调嵌套太多，流程复杂，难以驾驭。

不过，社区已经提供了很多方案，比如：Async,promise等流程化组件，就能很好的解决这个问题。具体细节，后面在阅读代码时再说。下面说说因为`回调`而更加棘手的问题——程序异常（错误）的处理。

3、异常要捕捉

回调太多、异常难捕捉，是Node.js被广为诟病的地方。Node.js是单进程的应用，异常如果未被正确处理，一旦抛出，就会引起整个进程死掉。而异常多发生在回调函数里，回调非常复杂的时候，异常很难定位。所以，很多人说，Node.js很快，但很脆弱。有利就有弊，这就是真实的世界。

与之形成鲜明对比的是，使用ruby on rails的小伙伴一定印象深刻，任何一个`error`发生，RoR抛出的错误信息里，都会明确给出出错的代码位置，而且通常都非常准确。

概念理解

（下面的内容，主要学习引用了这篇文章，NodeJS 错误处理最佳实践，我个人受益匪浅，感谢原作者)

异常，通常分类两种类型，一个是`失败`：正确的程序在运行时因为其他因素导致的失败，如：内存不足、网络不通、远程服务器塔机等，是可以预期的;另一个是`失误`，也就是bug，通常是程序员个人的问题，比如敲错了变量名、方法名，甚至逻辑错误等。

我们这里所讲的`异常`处理，就是第一类`失败`的情况，能够让正确的程序在任何情况下永远不`失败`，那才叫`健壮`。而对于另一种`失误`的处理，也就是Bug，只能要求程序员个人加强修炼，避免一些低级错误;对于一些深层次的逻辑错误，努力提高调试和测试水平，力争找到问题所在;或者发挥社区力量，群策群力。

失败类型

`失败`的原因通常是客观存在的，可以在代码里被有效处理。比如：系统本身（内存不足或者打开文件数过多），系统配置（没有到达远程主机的路由），网络问题（端口挂起），远程服务（500错误，连接失败）。主要场景，如：

- 连接不到服务器
- 无法解析主机名
- 无效的用户输入
- 请求超时
- 服务器返回500
- 套接字被挂起
- 系统内存不足

`失误`的原因多为主观因素，除非修正错误，否则永远无法被有效处理。主要场景，如：

- 读取一个 undefined 属性
- 调用异步函数没有指定回调
- 传递了错误参数：该传对象的时候传了一个字符串，或其他内容等

处理方法

错误处理并不是可以凭空加到一个没有任何错误处理的程序中的。没有办法在一个集中的地方处理所有的异常。需要考虑任何会导致失败的代码（比如打开文件，连接服务器，Fork子进程等）可能产生的结果。包括为什么出错，错误背后的原因。

更具体点说。对于一个给定的错误，可以做这些事情:

（**1**）直接处理。有的时候该做什么很清楚。如果你在尝试打开日志文件的时候得到了一个ENOENT错误，很有可能你是第一次打开这个文件，你要做的就是首先创建它。更有意思的例子是，你维护着到服务器（比如数据库）的持久连接，然后遇到了一个“socket hang-up”的异常。这通常意味着要么远端要么本地的网络失败了。很多时候这种错误是暂时的，所以大部分情况下你得重新连接来解决问题。（这和接下来的重试不大一样，因为在你得到这个错误的时候不一定有操作正在进行）

（**2**）报告给客户端。如果你不知道怎么处理这个异常，最简单的方式就是放弃你正在执行的操作，清理所有开始的，然后把错误传递给客户端。这种方式适合错误短时间内无法解决的情形。比如，用户提交了不正确的JSON，你再解析一次是没什么帮助的。

（**3**）重试操作。对于那些来自网络和远程服务的错误，有的时候重试操作就可以解决问题。比如，远程服务返回了503（服务不可用错误），你可能会在几秒种后重试。如果确定要重试，你应该清晰的用文档记录下将会多次重试，重试多少次直到失败，以及两次重试的间隔。另外，不要每次都假设需要重试。如果在栈中很深的地方（比如，被一个客户端调用，

而那个客户端被另外一个由用户操作的客户端控制），这种情形下快速失败让客户端去重试会更好。如果栈中的每一层都觉得需要重试，用户最终会等待更长的时间，因为每一层都没有意识到下层同时也在尝试。

（**4**）直接崩溃。对于那些本不可能发生的错误，或者由程序员失误导致的错误（比如无法连接到同一程序里的本地套接字），可以记录一个错误日志然后直接崩溃。其它的比如内存不足这种错误，是JavaScript这样的脚本语言无法处理的，崩溃是十分合理的。（即便如此，在child_process.exec这样的分离的操作里，得到ENOMEM错误，或者那些你可以合理处理的错误时，你应该考虑这么做）。在你无计可施需要让管理员做修复的时候，你也可以直接崩溃。如果你用光了所有的文件描述符或者没有访问配置文件的权限，这种情况下你什么都做不了，只能等某个用户登录系统把东西修好。

（**5**）记录错误。有的时候你什么都做不了，没有操作可以重试或者放弃，没有任何理由崩溃掉应用程序。举个例子，你用DNS跟踪了一组远程服务，结果有一个DNS失败了。除了记录一条日志并且继续使用剩下的服务以外，你什么都做不了。但是，你至少得记录点什么（凡事都有例外。如果这种情况每秒发生几千次，而你又没法处理，那每次发生都记录可能就不值得了，但是要周期性的记录）。

编码实践

（**1**）错误的编码方式

下面的代码是不能正确处理异常的。如果不明白，请重新温习一下 `回调` 部分，记住“事事皆回调”。

```
function myApiFunc(callback)
{
    /*
     * 这种模式并不工作
     */
    try {
      doSomeAsynchronousOperation(function (err) { //异步的原因，这里的回调函数被放在另一个线程独立工作，并不在try/catch下工作
        if (err)
          throw (err);
        /* continue as normal */
      });
    } catch (ex) {
      callback(ex);
    }
}
```

用惯了其他语言进行同步编程，一开始很容易这么写代码。直观上，上面的代码是想要在出现异步错误的时候调用callback，并把错误作为参数传递。他们错误地认为在自己的回调函数（传递给 doSomeAsynchronousOperation 的函数）里throw一个异常，会被外面的catch代码

块捕获。

try/catch和异步函数不是这么工作的。回忆一下，`回调`（异步）函数的意义就在于被调用的时候myApiFunc函数已经执行了（非柱塞），这意味着try代码块已经退出。这个回调函数外面，事实上，并没有try的代码块在作用。如果用这个模式，结果就是抛出异常的时候，程序崩溃了。

（**2**）怎么传递错误？

Node.js提供了3种基本的传递模式，分别是Throw， Callback，以及 EventEmitter：

- throw以同步的方式传递异常--也就是在函数被调用处的相同的上下文。如果调用者（或者调用者的调用者）用了try/catch，则异常可以捕获。如果所有的调用者都没有用，那么程序通常情况下会崩溃（异常也可能会被domains或者进程级的uncaughtException捕捉到，详见下文）。

- Callback是最基础的异步传递事件的一种方式。用户传进来一个函数（callback），之后当某个异步操作完成后调用这个 callback。通常 callback 会以callback(err,result)的形式被调用，这种情况下， err和 result必然有一个是非空的，取决于操作是成功还是失败。

- 更复杂的情形是，函数没有用 Callback 而是返回一个 EventEmitter 对象，调用者需要监听这个对象的 error事件。这种方式在下面两种情况下很有用，

一种是：当在做一个可能会产生多个错误或多个结果的复杂操作的时候。比如，有一个请求一边从数据库取数据一边把数据发送回客户端，而不是等待所有的结果一起到达。在这个例子里，没有用 callback，而是返回了一个 EventEmitter，每个结果会触发一个row 事件，当所有结果发送完毕后会触发end事件，出现错误时会触发一个error事件。

另一种：用在那些具有复杂状态机的对象上，这些对象往往伴随着大量的异步事件。例如，一个套接字是一个 EventEmitter，它可能会触发“connect“，”end“，”timeout“，”drain“，”close“事件。这样，很自然地可以把”error“作为另外一种可以被触发的事件。在这种情况下，清楚知道”error“还有其它事件何时被触发很重要，同时被触发的还有什么事件（例如”close“），触发的顺序，还有套接字是否在结束的时候处于关闭状态。

其实，callback 和 EventEmitter 可以归为一类，不要同时使用。

（**3**）怎么使用它们？

通用的准则是：你即可以同步传递错误（抛出），也可以异步传递错误（通过传给一个回调函数或者触发EventEmitter的 error事件），但是不用同时使用。具体用哪一个取决于异常是怎么传递的，这点得在文档里说明清楚。

（**4**） `domain` 和 `process.on('uncaughtException')` 用还是不用？

客观上，`失败`可以被显示的机制所处理。`Domain` 以及进程级别的`uncaughtException`主要是用来从未料到的程序错误恢复的，这两种方式不鼓励使用。其实，它们通常被用在整个应用级别，作为一种保障机制，而不是在具体编码过程中。

代码示例

下面的函数会异步地连接到一个IPv4地址的TCP端口。

```
/*
* Make a TCP connection to the given IPv4 address.  Arguments:
*
*    ip4addr        a string representing a valid IPv4 address
*
*    tcpPort        a positive integer representing a valid TCP port
*
*    timeout        a positive integer denoting the number of milliseconds
*                   to wait for a response from the remote server before
*                   considering the connection to have failed.
*
*    callback       invoked when the connection succeeds or fails.  Upon
*                   success, callback is invoked as callback(null, socket),
*                   where `socket` is a Node net.Socket object.  Upon failure,
*                   callback is invoked as callback(err) instead.
*
* This function may fail for several reasons:
*
*    SystemError    For "connection refused" and "host unreachable" and other
*                   errors returned by the connect(2) system call.  For these
*                   errors, err.errno will be set to the actual errno symbolic
*                   name.
*
*    TimeoutError   Emitted if "timeout" milliseconds elapse without
*                   successfully completing the connection.
*
* All errors will have the conventional "remoteIp" and "remotePort" properties.
* After any error, any socket that was created will be closed.
*/
function connect(ip4addr, tcpPort, timeout, callback)
{
assert.equal(typeof (ip4addr), 'string',
    "argument 'ip4addr' must be a string");
assert.ok(net.isIPv4(ip4addr),
    "argument 'ip4addr' must be a valid IPv4 address");
assert.equal(typeof (tcpPort), 'number',
    "argument 'tcpPort' must be a number");
assert.ok(!isNaN(tcpPort) && tcpPort > 0 && tcpPort < 65536,
    "argument 'tcpPort' must be a positive integer between 1 and 65535");
assert.equal(typeof (timeout), 'number',
    "argument 'timeout' must be a number");
assert.ok(!isNaN(timeout) && timeout > 0,
    "argument 'timeout' must be a positive integer");
assert.equal(typeof (callback), 'function');

/* do work */
}
```

这个例子很简单，但是展示了上面所谈论的一些建议：

- 参数，类型以及其它一些约束被清晰的文档化。

- 这个函数对于接受的参数是非常严格的，并且会在得到错误参数的时候抛出异常（程序员的失误）。
- 可能出现的`失败`集合被记录了。通过不同的"name"值可以区分不同的异常，而"errno"被用来获得系统错误的详细信息。
- 异常被传递的方式也被记录了（通过失败时调用回调函数）。
- 返回的错误有"remoteIp"和"remotePort"字段，这样用户就可以定义自己的错误了（比如，一个HTTP客户端的端口号是隐含的）。
- 虽然很明显，但是连接失败后的状态也被清晰的记录了：所有被打开的套接字此时已经被关闭。

Error 对象属性命名约定

强烈建议在发生错误的时候用这些名字来保持和Node.js核心以及Node.js插件的一致。这些大部分不会和某个给定的异常对应，但是出现疑问的时候，应该包含任何看起来有用的信息，即从编程上，也从自定义的错误消息上。

```
localHostname，localIp，localPort，remoteHostname，remoteIp，remotePort，path，srcpath，v
dstpath

hostname，ip，propertyName，propertyValue，syscall，errno
```

总结

本文讨论的内容，不仅是初学者经常困惑的地方，很多高手也会不自觉的掉进陷阱。理解了流，有效解决了回调和异常，编写Node.js程序就是一个简单且享受的过程。

本文原本想具体讨论Async、domain等非常具体的解决方案的，不过，限于篇幅和时间，只好暂时放在后面文章里说明。参考里，有几篇较好的文章，有兴趣的，自行翻阅吧。

截至本文，我们用4篇文章，首先考查了币圈Node.js的应用情况，简单介绍了Node.js入门知识，为研究学习Node.js应用奠定了基础。接下来，正式进入`源码解读`阶段，请看下一篇：亿书，一个面向未来的自出版平台，简单介绍之后，我们会从整体上分析它的功能模块布局。

参考

node.js中流(stream)的理解：http://segmentfault.com/a/1190000000519006

Linux管道PIPE的原理和应用: https://www.hitoy.org/pipe-aplication-in-linux.html

在单线程的情况下，NodeJs是如何分发子任务去执行的？
http://www.zhihu.com/question/24780826

Node is Not Single Threaded： http://rickgaribay.net/archive/2012/01/28/node-is-not-single-threaded.aspx

Node.js 异步异常的处理与domain模块解析: https://cnodejs.org/topic/516b64596d38277306407936

第三部分： 源码解读

我们的最终目标是要使用 `Node.js` 开发一款像比特币一样的加密货币。很显然，在开始之前，如果有现成的经验可以学习借鉴，就不用再去“重复制造轮子”，这是目前开发领域的重要 `共识` 。

站在巨人肩上，一方面可以快速了解 `Node.js` 技术知识，一方面可以对 `加密货币` 有一个更加深刻的学习理解，一举两得。这部分咱们就来学习一款这样的产品，具体的方法是：

1. 源码解读：掌握代码的功能，理清代码的运作流程。通常会提供详细的UML类图来说明。
2. 概念解析：功能自然都涉及到 `加密货币` 的机理，比如：什么是区块链，什么POS，如何加密解密等等，需要用写篇幅学习和补充。
3. 技术研究：针对代码用到的模块或第三方包，进行详细的解读。如果有时间，会通过写测试代码或开启类似 `sacdl-project` 的方式进行。

不同兴趣或技术水平的朋友，可以根据自己的情况选择性的阅读。比如，加密货币了解多的朋友，只要阅读1、3部分就是了，而技术水平高的小盆友只要阅读1、2就是了。没有兴趣的朋友，当然直接走人是最好的选择。

为了方便阅读，这部分是主要是 `源码解读` 。

我们分享的源码是：

网址： https://github.com/Ebookcoin/ebookcoin.git

版本： v0.1.3

该代码基于Crypti和宽松的MIT协议，尽管该项目已经无人维护，但是仍然感谢Crypti和它的原始团队原始团队为社区做出的贡献。

亿书，一个面向未来的自出版平台

前言

本篇的目的是通过一个产品的发行说明（白皮书），来了解产品最初的设计需求。无论是代码设计，还是源码分析，我们都要循着这些需求去渐次深入到代码内部，这样的思路，会帮助我们更加轻松、快速的掌握代码精髓。

亿书，是什么？

官方定义（白皮书 v1.0 2016.5.1）

> 亿书，英文名Ebook，是一个去中心化的出版平台，由新一代加密货币驱动，具备版权签名与认证、 协同创作、一键发布等功能，将促进人们更加主动地积累知识、分享经验，为人类创作注入新动力。 对于出版社等企业用户和第三方开发者，它提供了侧链功能，可以基于亿书强大的网络和市场， 使用亿书侧链、智能合约、云存储和计算节点，构建、发布个性化的去中心化软件，货币化一切有 形或无形资产，并从中盈利。

核心目标

亿书的核心目标是通过记录这个人类古老而简单的诉求，为知识创作和积累注入新动力，进而建立覆盖全人类的P2P网络，改善人类使用网络的体验，打造包括电子商务在内的融合社会化、信息化、商业化、物联网的全新网络。

使用场景

对普通人而言，亿书与日常使用的办公软件（word，wps等）相似，就是一款简单的文字写作工具，具备安装简单、编辑可视、互动协作等功能，还可直接获得海量专业、系统的电子书籍。

对于博客爱好者，它可以安装在服务器端，提供公开访问的能力，大大简化博客安装、个性化与维护的难度。对于专业作者，它的电子书编辑、一键发布、版权保护与交易等自出版功能，具备强大吸引力。

对于出版社等企业用户和第三方开发者，它提供了侧链功能，可以基于亿书强大的网络和市场，使用亿书侧链、智能合约、云存储和计算节点，构建、发布个性化的去中心化软件，货币化一切有形或无形资产，并从中盈利。

对于读者、作者和开发者而言，亿书就是一个知识宝库、巨大市场和一站式解决方案，是一个加密货币驱动的相互促进、互为所用、共享共赢的生态系统。

核心功能

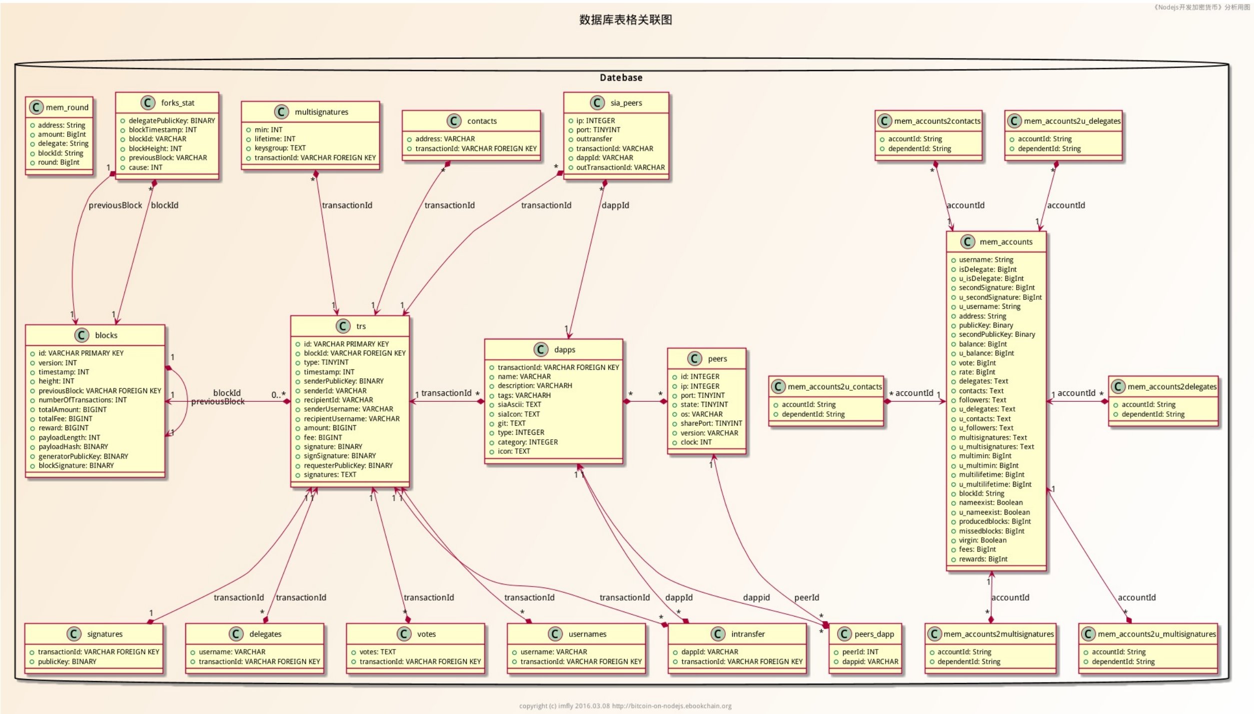
- 新一代极具创新精神的加密货币；
- 高性能对等网络；
- 去中心化的存储和计算；
- 易用易扩展的可编程侧链；
- 简单易用的可视化编辑器；
- 多重签名；
- 去中心化博客
- 自出版平台
- 版权签名与验证
- 针对主流开源产品的官方插件；
- 面向第三方开发者的开发工具包SDK。

技术架构

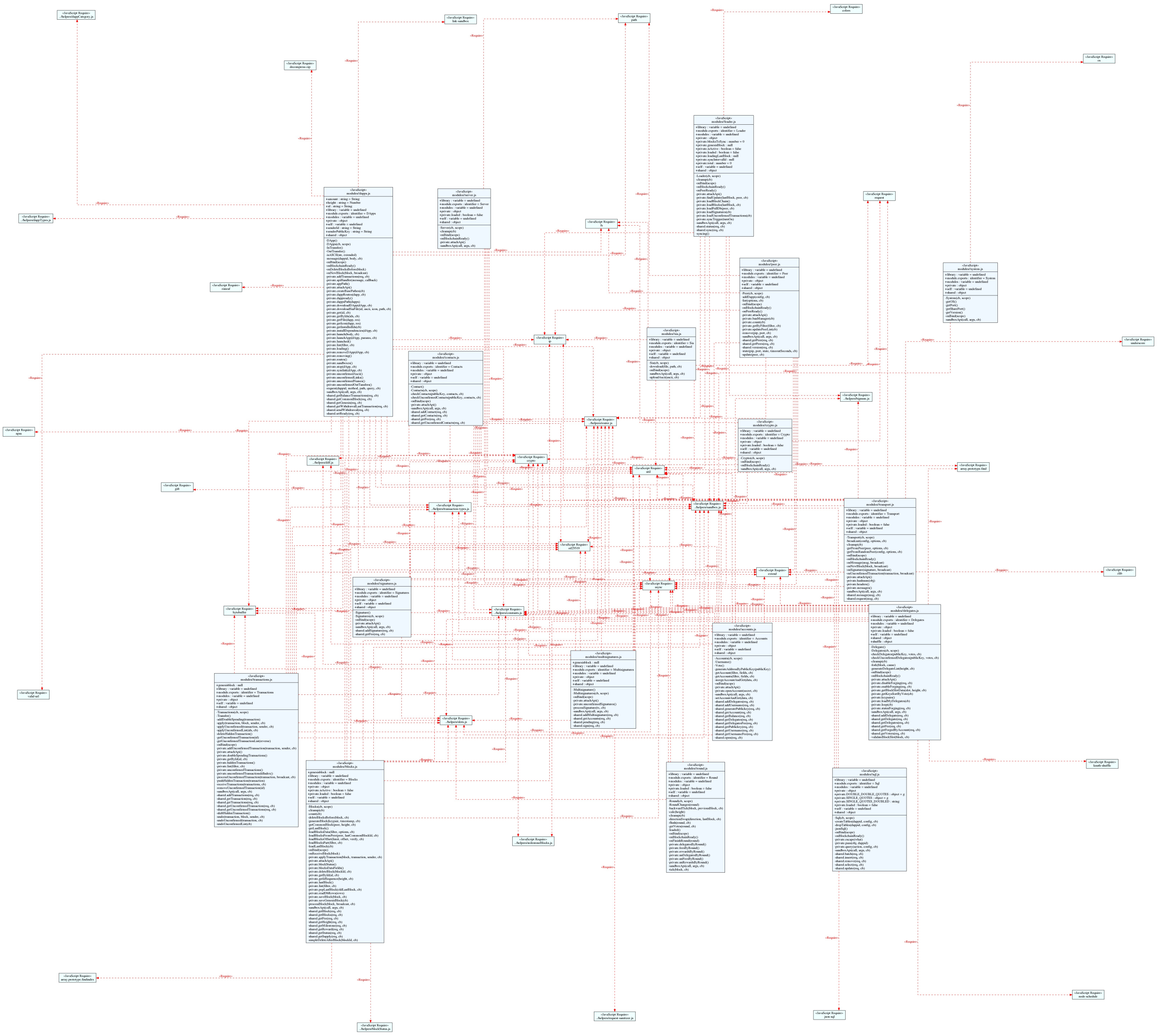
亿书完全基于Node.js平台研发，后台使用Express.js框架，前端使用Ember.js框架，客户端使用Electron框架，数据库使用SQLite，前后端统一使用Javascript脚本语言，界面使用HTML5和 CSS3。

亿书，使用建立在HTTP协议之上的点对点网络，基于DPOS（授权股权证明机制）共识算法，无需挖矿，大约1亿枚币子。每个块的时间为10秒，每个周期的101个区块均由 101个代表随机生成，广播并添加到区块链里，在得到6-10个确认后，交易完成，一个完整的101个块的周期大概需要16分钟。

数据库表格及其关联关系模型：



从代码上看，它使用Express作为开发框架，各功能模块都放在 `modules` 文件夹里，结构简单清晰。我们看看它核心模块的UML图，大概了解一下吧。



总结

本篇主要是概览，目的是了解我们要分享的东东是什么。同时也说明，该应用的价值：既可以学习加密货币开发，也可以对侧链的开发有所了解，是目前覆盖区块链技术多个方面的应用之一。

参考

亿书白皮书v1.0 http://ebookchain.org/ebookchain.pdf

亿书官方网站： http://ebookchain.org

入口程序app.js解读

前言

在入门文章部分，我们已经知道，Node.js的应用最终都可以合并成一个文件，为了开发方便，才将其拆分成多个文件。

被拆分的那个文件，自然是我们重点研究的对象，通常这个文件就是App.js或server.js，大家称之为 `入口程序` 。

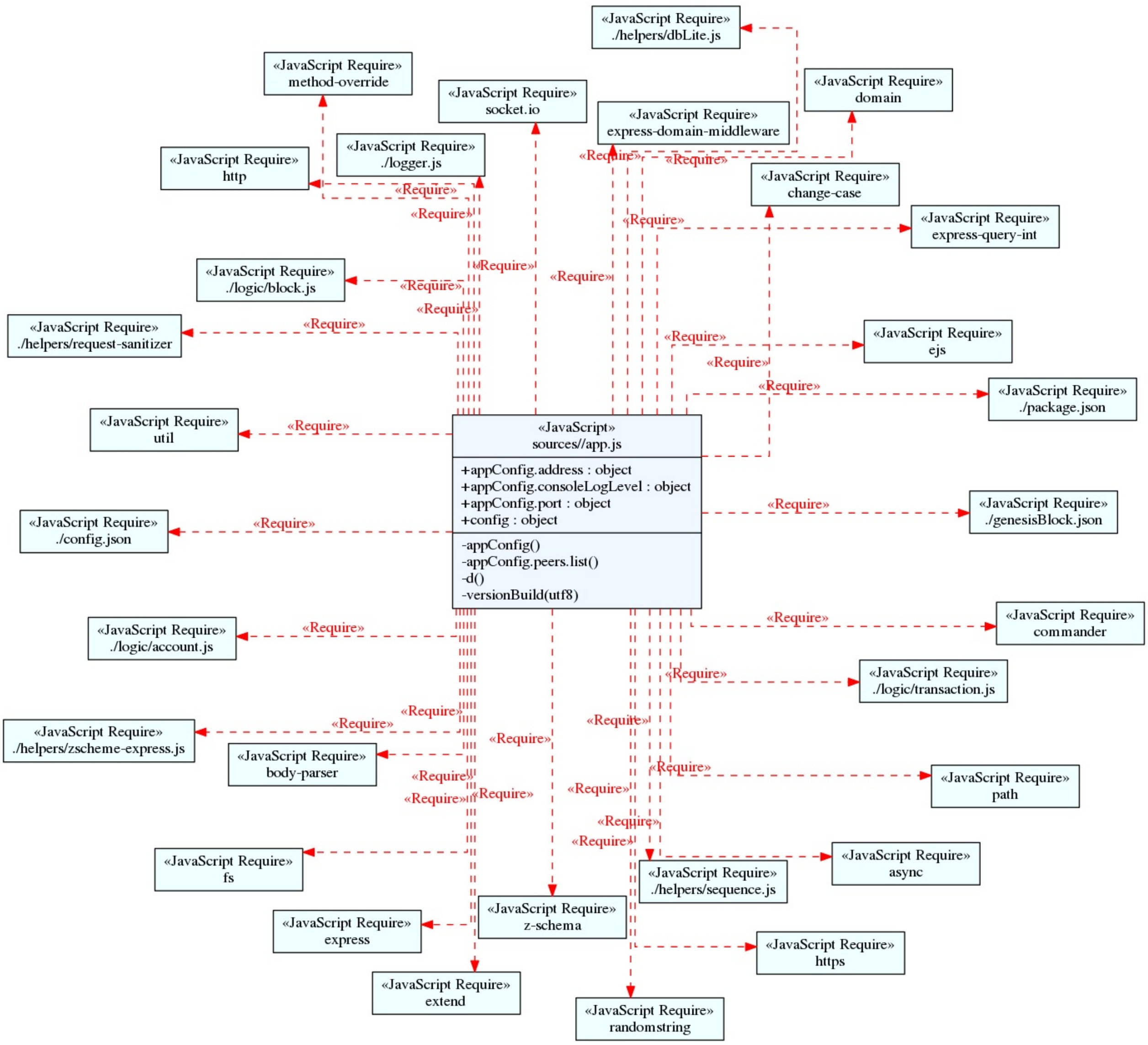
显然Ebookcoin用的就是app.js。这一篇，我们就来阅读一下该文件，学习研究它的整体架构流程。

源码

地址： https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/app.js

类图

js原本无类，因此它的类图并不好处理，仅能大致给出它与其他模块的关联关系。



解读

直接读代码看看。

1.配置处理

任何一个应用，都会提供一些参数。对这些参数的处理，有很多种方案。但总的来说，通常需要提供一种理想环境，即默认配置，同时给你一种方法自行修改。

（1）全局默认配置

通常默认参数较少时，可以硬编码到代码里。但更灵活的方式，就是使用单独文件。这里就使用了文件 `./config.json` 来保存全局配置，如：

```
{
    "port": 7000,
    "address": "0.0.0.0",
    "serveHttpAPI": true,
    "serveHttpWallet": true,
    "version": "0.1.1",
    "fileLogLevel": "info",
    "consoleLogLevel": "log",
    "sharePort": true,
    ...
```

使用时，只需要 `require` 就可以了。源码：

```
var appConfig = require("./config.json"); // app.js 4行
```

不过，为了灵活性，默认值通常允许用户修改。

（2）使用 `commander` 组件，引入命令行选项

`commander` 是Node.js第三方组件（使用npm安装），常被用来开发命令行工具，用法极为简单，详细内容请看开发实践部分的分享。源码：

```
// 1行
var program = require('commander');

// 19行
program
    .version(packageJson.version)
    .option('-c, --config <path>', 'Config file path')
    .option('-p, --port <port>', 'Listening port number')
    .option('-a, --address <ip>', 'Listening host name or ip')
    .option('-b, --blockchain <path>', 'Blockchain db path')
    .option('-x, --peers [peers...]', 'Peers list')
    .option('-l, --log <level>', 'Log level')
    .parse(process.argv);
```

这样，就可以在命令行执行命令时，加带 `-c`，`-p` 等选项，例如：

```
node app.js -p 8888
```

这时，该选项就以 `program.port` 的形式被保存，于是手动修改一下：

```
// 39行
if (program.port) {
    appConfig.port = program.port;
}
```

这是处理Node.js应用全局配置的一种常用且简单的方式，值得学习。

更多内容，我们在下一篇对 `commander` 组件进行详细介绍。

2.异常捕捉

我们在第一部分总结时，特意提到 `异常要捕捉` ，这里我们很轻松就可以看出来，代码对全局异常处理的方式。

注意：对于 `domain` 模块，已经不提倡使用，这部分代码将再后续的更新中去除，这里仅做了解就是了。

（1）使用 `uncaughtException` 捕捉进程异常

```
// 65行
process.on('uncaughtException', function (err) {
    // handle the error safely
    logger.fatal('System error', { message: err.message, stack: err.stack });
    process.emit('cleanup');
});
```

（2）使用 `domain` 模块捕获全局异常

```
// 96行
var d = require('domain').create();
d.on('error', function (err) {
    logger.fatal('Domain master', { message: err.message, stack: err.stack });
    process.exit(0);
});
d.run(function () {
...
```

另外，对各个模块，也使用了 `domain`

```
// 415行
var d = require('domain').create();

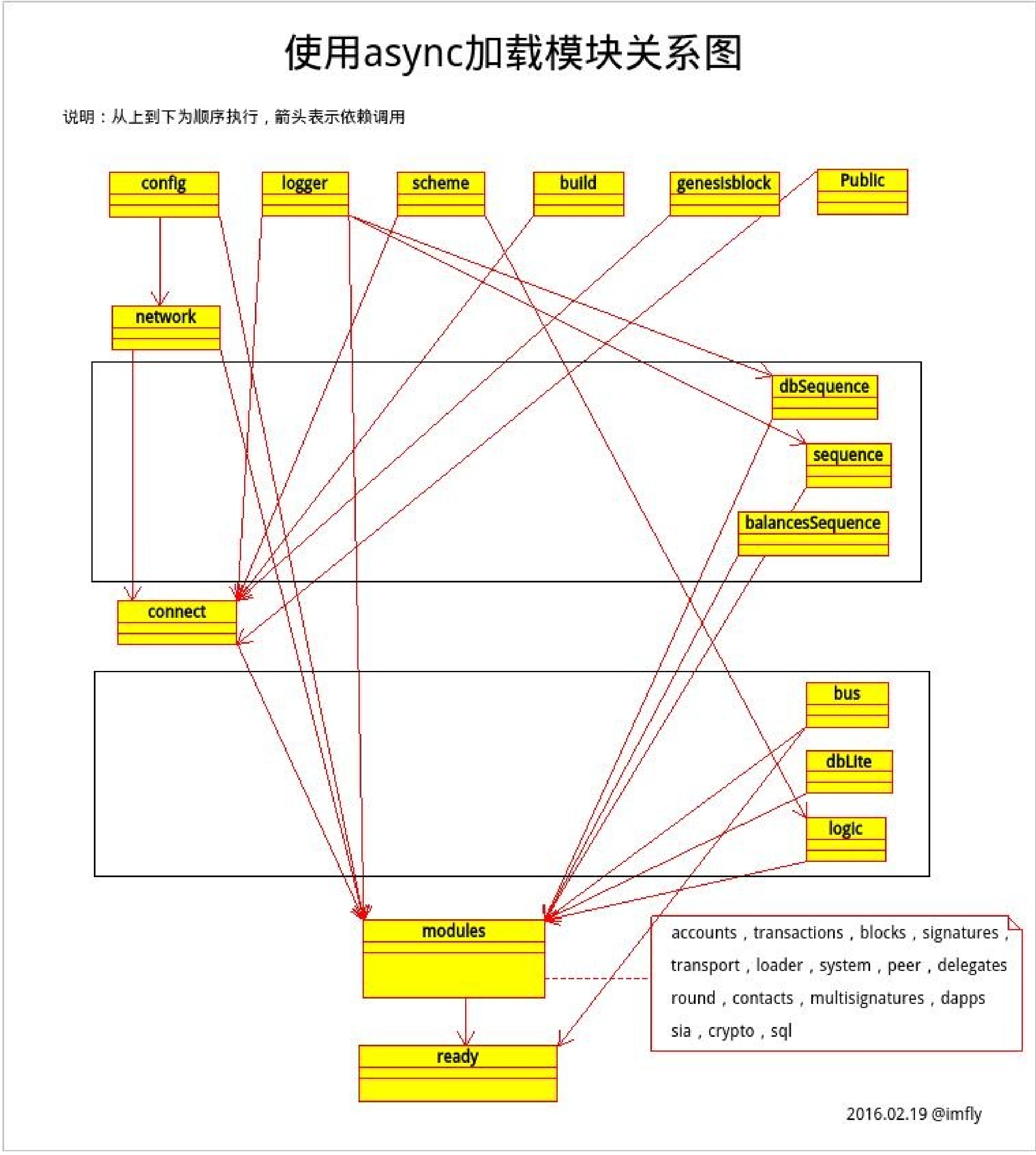
d.on('error', function (err) {
scope.logger.fatal('domain ' + name, {message: err.message, stack: err.stack});
});
...
```

3.模块加载

这才是真正的重点，不过看过代码，发现一切都那么干净利略，也没有多层`回调`那些`大坑`，原来是用了`async`流程管理组件。

整体使用`async.auto`进行`顺序调用`；在加载`modules`时，又使用`async.parallel`，使其并行运作;当发生错误时，清理工作用到了`async.eachSeries`。

下图是手工简单画的，说明了从代码103-438行之间，各模块的加载运行顺序。



下篇，我们也有必要对 `async` 组件进行详解梳理。这里，您只要能猜出代码意图，就不用太操心async的用法。下面，读读有关源码：

（1）初始网络

我们从 `packages.json` 里看到使用了 `Express` 框架。通过前面部分的介绍，知道必须在入口程序里，初始化才对。具体如何调用的？下面的代码，显然十分熟悉。

我们知道，Express是Node.js重要的web开发框架，这里的网络 `network` 本质上就是以Express为基础的web应用，自然 `白皮书` 才会宣扬 `基于Http协议` 。

```
// 215行
network: ['config', function (cb, scope) {
    var express = require('express');
    var app = express();
    var server = require('http').createServer(app);
    var io = require('socket.io')(server);

    if (scope.config.ssl.enabled) {
        var privateKey = fs.readFileSync(scope.config.ssl.options.key);
        var certificate = fs.readFileSync(scope.config.ssl.options.cert);

        var https = require('https').createServer({
    ...
```

说明：这是 `async.auto` 常用的方法， `network` 用到的任何需要回调的方法（这里是 `config` ），都放在这个数组里，最后的回调函数(`function (cb, scope) {//code}`)，可以巧妙的调用，如： `scope.config` 。

这里的代码，仅仅初始化服务，没有做太多实质的事情，真正的动作在下面。

（2）构建链接

从下面的代码开始，我们才能看到这个应用的本质。270行代码用到了 `network` 等，如下：

```
// 270行
connect: ['config', 'public', 'genesisblock', 'logger', 'build', 'network', function (
cb, scope) {
```

接着，下面的代码，加载了几个中间件，告诉我们，该应用接受 `ejs` 模板驱动的html文件，视图文件和图片、样式等静态文件都在 `public` 文件夹等，这些信息绝对比官方文档还有用。

```
// 277行
scope.network.app.engine('html', require('ejs').renderFile);
scope.network.app.use(require('express-domain-middleware'));
scope.network.app.set('view engine', 'ejs');
scope.network.app.set('views', path.join(__dirname, 'public'));
scope.network.app.use(scope.network.express.static(path.join(__dirname, 'public')));
...
```

再下来，就是对请求参数和响应数据的处理，包括对节点 `peers` 中黑名单、白名单的过滤等，最后启动服务操作：

```
// 336行
scope.network.server.listen(scope.config.port, scope.config.address, function (err) {
```

（3）加载逻辑

看代码知道，其核心逻辑功能是：账户管理、交易和区块链。这些模块，本质上是对数据库操作的封装，`account` 与 `modules/accounts` 模块对应，`transaction` 与 `modules/transactions` 模块对应，`block` 与 `modules/blocks` 模块对应，我们会在介绍相关模块时，一起分析。

```
// 379行
logic: ['dbLite', 'bus', 'scheme', 'genesisblock', function (cb, scope) {

    // 嵌套了async.auto
    async.auto({
        ...
        account: ["dbLite", "bus", "scheme", 'genesisblock', function (cb, scope) {
            new Account(scope, cb);
        }],
        transaction: ["dbLite", "bus", "scheme", 'genesisblock', "account", function (
cb, scope) {
            new Transaction(scope, cb);
        }],
        block: ["dbLite", "bus", "scheme", 'genesisblock', "account", "transaction", f
unction (cb, scope) {
            new Block(scope, cb);
        }]
    }, cb);
    ...
```

（4）加载模块

上面所有代码的执行结果，都要被这里的各模块共享。下面的代码说明，各个模块都采用一致（不一定一样）的参数和处理方法，这样处理起来简单方便：

```
// 411行
modules: ['network', 'connect', 'config', 'logger', 'bus', 'sequence', 'dbSequence', '
balancesSequence', 'dbLite', 'logic', function (cb, scope) {

    // 对每个模块都使用`domain`监控其错误
    Object.keys(config.modules).forEach(function (name) {
        tasks[name] = function (cb) {
            var d = require('domain').create();

            d.on('error', function (err) {
                ...
            });

            d.run(function () {
                ...
            });
        }
    });

    // 让各个模块并行运行
    async.parallel(tasks, function (err, results) {
        cb(err, results);
    });
```

这里的模块既然都是并行处理，研究它们就不需要分先后了。

总结

这篇文章总算深入代码了，但是仍然较为粗略。不过，对整个应用的基本架构已经了然。继续深入研究，方向路线也已然清晰。

代码中还有很多细节，我们并没有逐行介绍。个人认为，读代码就像看文章，先要概览，逐步深入，不一定一开始就逐字逐句去读，那样效率低、效果差。

对于这个app.js文件，成手读它可能就是分分钟的事情，而写出来却要罗嗦这么多。如果，你并没有觉得很轻松，甚至理解很困难，那么可能缺少对 `commander` 、 `domain` 和 `async` 等组件或模块的了解，请看下一篇：《**Node.js**开发加密货币》之七：技术研究**——commander**、**domain**和**async**介绍

一个精巧的p2p网络实现

前言

加密货币都是去中心化的应用，去中心化的基础就是P2P网络，其作用和地位不言而喻，无可替代。当然，对于一个不开源的所谓私链（私有区块链），是否必要，尚无定论。

事实上，P2P网络不是什么新技术。但是，使用Node.js开发的P2P网络，确实值得围观。这一篇，我们就来看看Ebookcoin的点对点网络是如何实现的。

源码

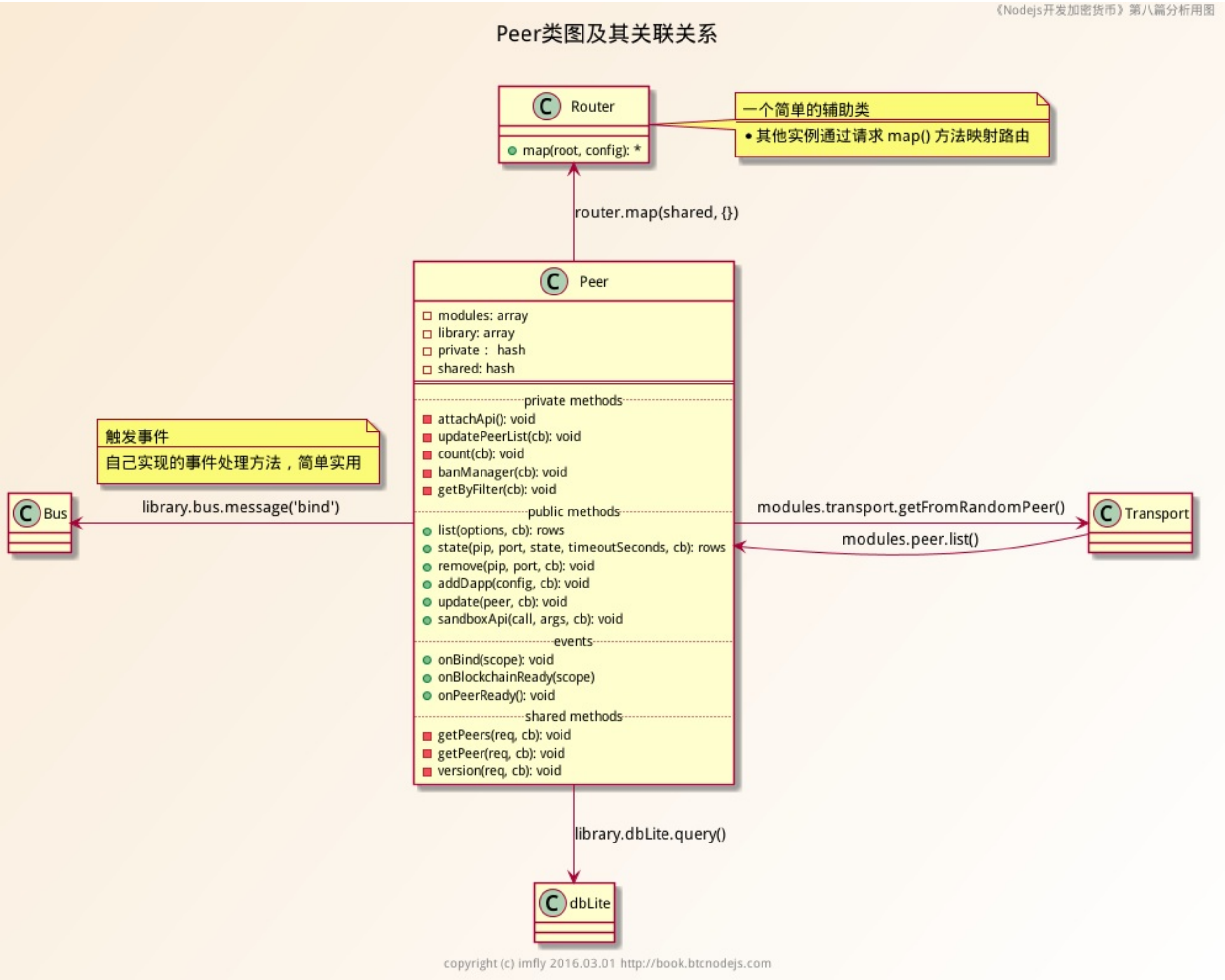
主要源码地址：

peer.js: https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/peer.js

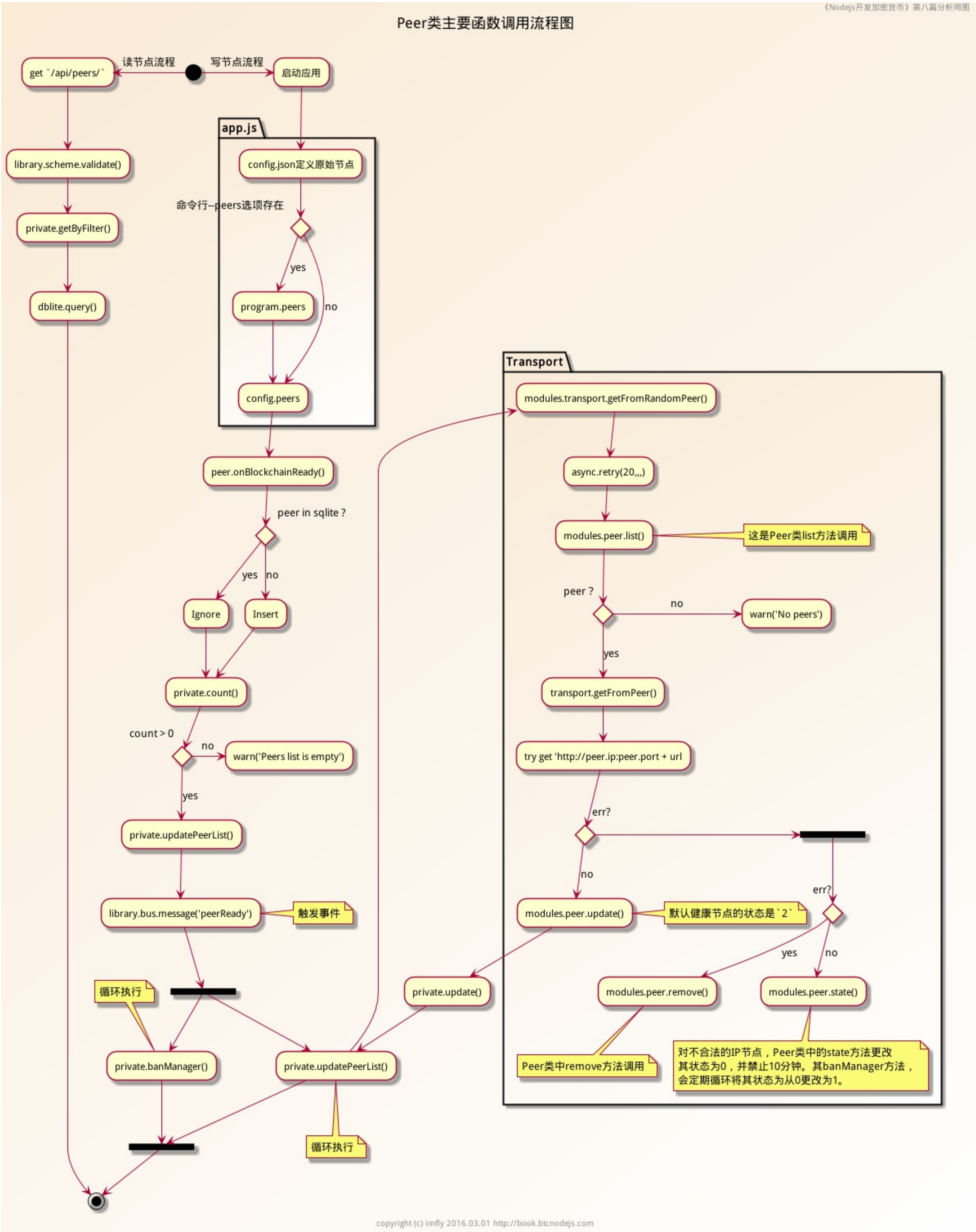
transport.js: https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/transport.js

router.js: https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/helpers/router.js

类图


《Nodejs开发加密货币》第八篇分析用图
Peer类图及其关联关系
Router
map(root, config): *
一个简单的辅助类
其他实例通过请求 map() 方法映射路由
router.map(shared, {})
Peer
modules: array
library: array
private : hash
shared: hash
private methods
attachApi(): void
updatePeerList(cb): void
count(cb): void
banManager(cb): void
getByFilter(cb): void
public methods
list(options, cb): rows
state(pip, port, state, timeoutSeconds, cb): rows
remove(pip, port, cb): void
addDapp(config, cb): void
update(peer, cb): void
sandboxApi(call, args, cb): void
events
onBind(scope): void
onBlockchainReady(scope)
onPeerReady(): void
shared methods
getPeers(req, cb): void
getPeer(req, cb): void
version(req, cb): void
触发事件
自己实现的事件处理方法，简单实用
Bus
library.bus.message('bind')
modules.transport.getFromRandomPeer()
modules.peer.list()
Transport
library.dbLite.query()
dbLite
copyright (c) imfly 2016.03.01 http://book.btcnodejs.com

流程图



解读

基于http的web应用，抓住路由的定义、设计与实现，是快速弄清业务逻辑的简单方法。目前，分析的是 `modules` 文件夹下的各个模块文件，这些模块基本都是独立的Express微应用，在开发和设计上相互独立，各不冲突，逻辑清晰，这为学习分析，提供了便利。

1.路由扩展

任何应用，只要提供Web访问能力或第三方访问的Api，都需要提供从地址到逻辑的请求分发功能，这就是路由。Ebookcoin是基于http协议的Express应用，Express底层基于Node.js的connect模块，因此其路由设计简单而灵活。

前面，在入门部分，已经讲到对路由的分析调用，这里是其简单实现。先看看 `helper/router.js` 吧。

```
// 27行
var Router = function () {
    var router = require('express').Router();

    router.use(function (req, res, next) {
        res.header("Access-Control-Allow-Origin", "*");
        res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
        next();
    });

    router.map = map;

    return router;
}
...
```

这段代码定义了一个Express路由器 `Router` ，并扩展了两个功能：

- 允许任何客户端调用。其实，就是设置了跨域请求，选项 `Access-Control-Allow-Origin` 设置为 `*` ，自然任何IP和端口的节点都可以访问和被访问。
- 添加了地址映射方法。该方法的主要内容如下：

```
// 3行
function map(root, config) {
    var router = this;
    Object.keys(config).forEach(function (params) {
        var route = params.split(" ");
        if (route.length != 2 || ["post", "get", "put"].indexOf(route[0]) == -1) {
            throw Error("wrong map config");
        }
        router[route[0]](route[1], function (req, res, next) {
            root[config[params]]({"body": route[0] == "get" ? req.query : req.body}, f
unction (err, response) {
                if (err) {
                    res.json({"success": false, "error": err});
                } else {
                    return res.json(extend({}, {"success": true}, response));
                }
            });
        });
    });
}
```

该方法，接受两个对象作为参数：

- root: 定义了所要开放Api的逻辑函数;
- config: 定义了路由和root定义的函数的对应关系。

其运行的结果，就相当于：

```
router.get('/peers', function(req, res, next){
    root.getPeers(...);
})
```

这里关键的小技巧是，在js代码中，对象也是hash值，root.getPeers() 与 root'getPeers' 是一致的。不过后者可以用字符串变量代替，更加灵活，有点像ruby里的元编程。这是脚本语言的优势（简单的字符串拼接处理）。

扩展一下，在类似sails的框架（基于express)里，很多都是可以使用类似 `config.json` 的文件直接配置的，包括路由。参考这个函数，很容易理解和实现。

2.节点路由

很轻松就能在 `peer.js` 里找到上述map方法的使用：

```
// 3行
Router = require('../helpers/router.js')

// 25
privated.attachApi = function () {
    var router = new Router();

    router.use(function (req, res, next) {
        if (modules) return next();
        res.status(500).send({success: false, error: "Blockchain is loading"});
    });

    // 34行
    router.map(shared, {
        "get /": "getPeers",
        "get /version": "version",
        "get /get": "getPeer"
    });

    router.use(function (req, res) {
        res.status(500).send({success: false, error: "API endpoint not found"});
    });

  // 44行
    library.network.app.use('/api/peers', router);
    library.network.app.use(function (err, req, res, next) {
        if (!err) return next();
        library.logger.error(req.url, err.toString());
        res.status(500).send({success: false, error: err.toString()});
    });
};
```

上面代码的34行，可以直观想象到，会有类似 `/version` 的路由出现，44行是express应用，这里就是将定义好的路由放在 `/api/peers` 前缀之下，可以确信 `peer.js` 文件提供了下面3个公共Api地址：

> http://ip:port/api/peers/
>
> http://ip:port/api/peers/version
>
> http://ip:port/api/peers/get

当然，是不是可以直接这么调用，要看具体对应的函数是否还有其他的参数要求，比如：`/api/peers/get`，按照restful的api设计原则，可以理解为是获得具体某个节点信息，那么总该给个 `id` 之类的限定条件吧。看源码：

```
// 455行
library.scheme.validate(query, {
        type: "object",
        properties: {
            ip_str: {
                type: "string",
                minLength: 1
            },
            port: {
                type: "integer",
                minimum: 0,
                maximum: 65535
            }
        },
        required: ['ip_str', 'port']
    }, function (err) {
        ...
        // 480行
        privated.getByFilter({
            ...
        });
    });
```

这里，在具体运行过程中，library就是 `app.js` 里传过来的 `scope` ，该参数包含的scheme代表了一个 `z_schema` 实例。

`z_schema` 是一个第三方组件，具体请看参考链接。该组件提供了json数据格式验证功能。上述代码的意思是：对请求参数 `query` 进行验证，验证规则是：object类型，属性 `ip_str` 要求长度不小于1的字符串，属性 `port` 要求0~65535之间的整数，并且都不能空（必需）。

这就说明，我们应该这样请求 `http://ip:port/api/peers/get?ip_str=0.0.0.0&port=1234` ，不然会返回错误信息。回头看看 `getPeers` 方法的实现，没有 `required` 字段，对应可以直接访问 `http://ip:port/api/peers/` 。

看480行，上面的地址，都会调用 `privated.getByFilter()` ，并由它从sqlite数据库里查询数据表 `peers` 。这里涉及到 `dblite` 第三方组件 (请看参考链接)，对请求操作sqlite数据库进行了简单封装。

3.节点保存

大多数应用，读数据相对简单，难在写数据。上面的代码，都是 `get` 请求，可以查寻节点及其信息。我们自然会问，查询的信息从哪里来？初始的节点在哪里？节点变更了，怎么办？

（1）初始化节点

从现实角度考虑，在一个P2P网络中，一个孤立的节点，在没有其他任何节点信息的情况下，仅仅靠网络扫描去寻找其他节点，将是一件很难完成的事情，更别提高效和安全了。

因此，在运行软件之前，初始化一些节点供联网使用，是最简单直接的解决方案。这个在配置文件 `config.json` 里，有直接体现：

```
// config.json 15行
"peers": {
        "list": [],
        "blackList": [],
        "options": {
                "timeout": 4000
        }
},
...
```

list的数据格式为：

```
[
    {
        ip: 0.0.0.0,
        port: 7000
    },
    ...
]
```

当然，也可以在启动的时候，通过参数 `--peers 1.2.3.4:70001, 2.1.2.3:7002` 提供（代码见 `app.js` 47行）。

（**2**）写入节点

写入节点，就是持久化，或者保存到数据库，或者保存到某个文件。这里保存到sqlite3数据库里的 `peers` 表了，代码如下：

```
// peer.js 347行
Peer.prototype.onBlockchainReady = function () {
    async.eachSeries(library.config.peers.list, function (peer, cb) {
        library.dbLite.query("INSERT OR IGNORE INTO peers(ip, port, state, sharePort)
VALUES($ip, $port, $state, $sharePort)", {
            ip: ip.toLong(peer.ip),
            port: peer.port,
            state: 2, //初始状态为2，都是健康的节点
            sharePort: Number(true)
        }, cb);
    }, function (err) {
        if (err) {
            library.logger.error('onBlockchainReady', err);
        }

        privated.count(function (err, count) {
            if (count) {
                privated.updatePeerList(function (err) {
                    err && library.logger.error('updatePeerList', err);
                    // 364行
                    library.bus.message('peerReady');
                })
                library.logger.info('Peers ready, stored ' + count);
            } else {
                library.logger.warn('Peers list is empty');
            }
        });
    });
}
```

这段代码的意思是，当区块链（后面篇章分析）加载完毕的时候（触发事件），依次将配置的节点写入数据库，如果数据库已经存在相同的记录就忽略，然后更新节点列表，触发节点加载完毕事件。

这里对数据库 `Sqlite` 的插入操作，插入语句是 `library.dbLite.query("INSERT OR IGNORE INTO peers` ，有意思的是 `IGNORE` 操作字符串，是sqlite3支持的（见参考），当数据库有相同记录的时候，该记录被忽略，继续往下执行。

执行成功，就会调用 `library.bus.message('peerReady')` ，进而触发 `peerReady` 事件。该事件的功能就是：

（**3**）更新节点

事件 `onPeerReady` 函数，如下：

```
// peer.js 374行
Peer.prototype.onPeerReady = function () {
    setImmediate(function nextUpdatePeerList() {
        privated.updatePeerList(function (err) {
            err && library.logger.error('updatePeerList timer', err);
            setTimeout(nextUpdatePeerList, 60 * 1000);
        })
    });

    setImmediate(function nextBanManager() {
        privated.banManager(function (err) {
            err && library.logger.error('banManager timer', err);
            setTimeout(nextBanManager, 65 * 1000)
        });
    });
}
```

两个 `setImmediate` 函数的调用，一个循环更新节点列表，一个循环更新节点状态。

第一个循环调用

看看第一个循环调用的函数 `updatePeerList` ，

```
privated.updatePeerList = function (cb) {
    // 53行
    modules.transport.getFromRandomPeer({
        api: '/list',
        method: 'GET'
    }, function (err, data) {
        ...
        library.scheme.validate(data.body, {
                    ...
                    // 124行
                    self.update(peer, cb);
                });
            }, cb);
        });
    });
};
```

看53行，我们知道，程序通过 `transport` 模块的 `.getFromRandomPeer` 方法，逐个随机的验证节点信息，并将其做删除和更新处理。如此一来，各种调用关系更加清晰，看流程图更加直观。 `.getFromRandomPeer` 的代码：

```
// transport.js 474行
Transport.prototype.getFromRandomPeer = function (config, options, cb) {
    ...

    // 481行
    async.retry(20, function (cb) {
        modules.peer.list(config, function (err, peers) {
            if (!err && peers.length) {
                var peer = peers[0];

                // 485行
                self.getFromPeer(peer, options, cb);
            } else {
                return cb(err || "No peers in db");
            }
        });
   ...
};
```

代码很简单，重要的是理解 `async.retry` 的用法（下篇技术分享，详细学习），该方法就是要重复调用第一个task函数20次，有正确返回结果就传给回调函数。这里，只要查到一个节点，就会传给485行的 `getFromPeer` 函数，该函数是检验处理现存节点的核心函数，代码如下：

```
// transport.js 500行
Transport.prototype.getFromPeer = function (peer, options, cb) {
    ...
    var req = {
        // 519行： 获得节点地址
        url: 'http://' + ip.fromLong(peer.ip) + ':' + peer.port + url,
        ...
    };

    // 532行： 使用`request`组件发送请求
    return request(req, function (err, response, body) {
        if (err || response.statusCode != 200) {
            ...
            if (peer) {
                if (err && (err.code == "ETIMEDOUT" || err.code == "ESOCKETTIMEDOUT" |
| err.code == "ECONNREFUSED")) {

                    // 542行： 对于无法请求的，自然要删除
                    modules.peer.remove(peer.ip, peer.port, function (err) {
                    ...
                    });
                } else {
                    if (!options.not_ban) {

                        // 549行： 对于状态码不是200的，比如304等禁止状态，就要更改其状态
                        modules.peer.state(peer.ip, peer.port, 0, 600, function (err)
{
                        ...
                        });
                    }
                }
            }
            cb && cb(err || ('request status code' + response.statusCode));
            return;
        }

        ...
        if (port > 0 && port <= 65535 && response.headers['version'] == library.config
.version) {
            // 595行： 一切问题都不存在
            modules.peer.update({
                ip: peer.ip,
                port: port,
                state: 2, // 598行： 看来健康的节点状态为2
                ...
    });
}
```

这里最重要的是532行， `request` 第三方组件的使用，请看参考链接。官方说 `request` 为简单的http客户端，功能足够强大，可以模拟浏览器访问信息，经常被用来做测试。

第二个循环调用

第二个循环调用的函数很简单，就是循环更改 `state` 和 `clock` 字段，主要是将禁止的状态 `state=0` ，修改为 `1` ，如下：

```
// 142行
privated.banManager = function (cb) {
    library.dbLite.query("UPDATE peers SET state = 1, clock = null where (state = 0 an
d clock - $now < 0)", {now: Date.now()}, cb);
}
```

综上，整个P2P网络的读写和更新都已经清楚，回头再看活动图和类图，就更加明朗了。

最后，补充一下数据库里，节点表格 `peers` 的字段信息：
id,ip,port,state,os,sharePort,version,clock

总结

本篇，重点阅读了 `peer.js` 文件，学习了一个使用Node.js开发的P2P网络架构，其特点是：

- 产品提供初始节点列表，保障了初始化节点快速完成，不至于成为孤立节点;
- 节点具备跨域访问能力，任何节点之间都可以自由访问;
- 节点具备自我更新能力，定期查询和更新死掉的节点，保障网络始终畅通;

一旦达到一定的节点数量，就会形成一个互联互通的 `不死网络` 。搭建在这种网络上的服务，会充满怎样的诱惑？加密货币为什么会被认为是下一代互联网？这加起来不足千行的代码，可以给我们足够多的遐想空间。

这部分代码，涉及到 `dblite，request，z_schema` 等第三方组件，以及Ebookcoin自行实现的事件处理方法 `library.bus` (在 `app.js` 文件的行)，都很简单，不再分享或赘述，请自行查阅。本篇涉及的代码中，关于回调的设计很多，值得总结和研究。 `async` 组件，被反复使用，有必须汇总一下，请关注后续的技术分享。

参考

z_schema组件: https://github.com/Ebookcoin/z_schema

dblite组件： https://github.com/Ebookcoin/dblite

request组件： http://github.com/request/request

SQL As Understood By SQLite： https://www.sqlite.org/lang_conflict.html

加密和验证

前言

加密解密技术在加密货币开发中的作用不言而喻。但技术本身并不是什么新鲜事，重要的是如果没有前面的P2P网络，和后面要介绍的区块链，单独的加解密显然没有那么神奇，加密货币也不会成为无需验证、高度可信的强大网络。

但是，提到加解密技术，业界的通则是，使用现成的组件，严格按照文档去做，别自作聪明，这也是使用加密解密技术的最安全方式。这篇就来研究亿书是如何使用加解密技术的。

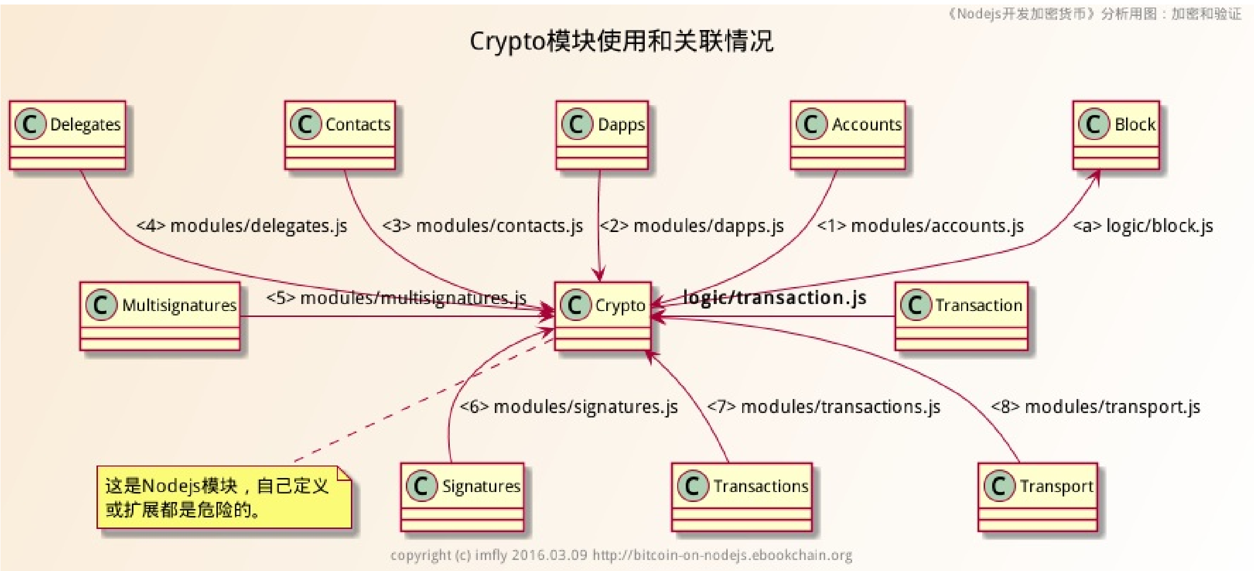
源码

`Ebookcoin` 没有提供相关扩展，全部使用Node.js自己的 `crypto` 模块进行加密，使用 `Ed25519` 组件签名认证。本文涉及到的代码：

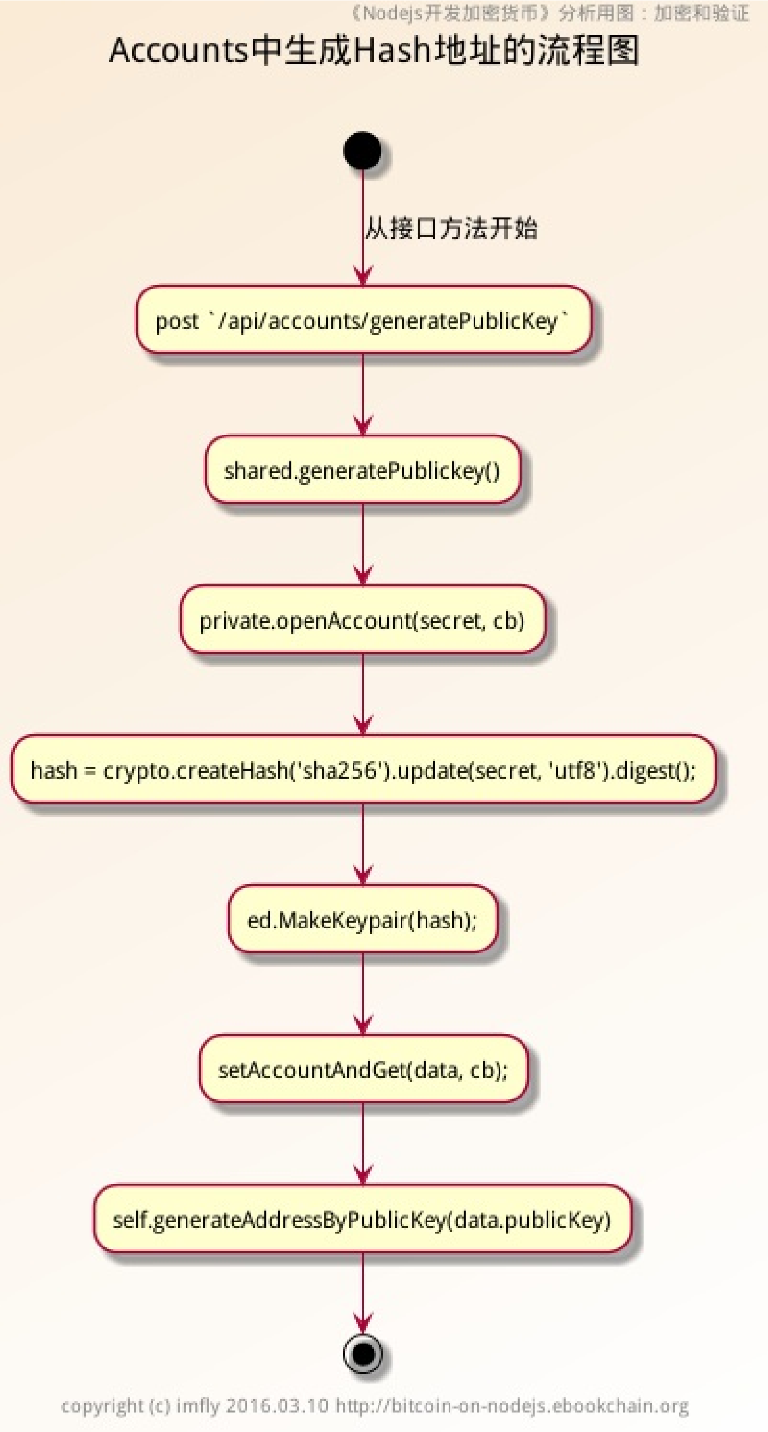
accounts.js: https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/accounts.js

account.js: https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/logic/account.js

类图



流程图



概念

仅仅介绍涉及到的最少概念（说实话，多了咱也不会，甚至不敢）。

(1)私钥和公钥

加密技术涉及的概念晦涩，讲个小故事，就一下清楚了。大学一哥们追女朋友有贼心没贼胆，一直不敢当面说“I love you”，就想了一招，顺手写下"J mpwf zpv"交给了另一位女生，让她帮忙传信。然后，等女朋友好奇打来电话时，他就告诉她依次向前顺延1个字母，组合起来

就是他想说的话。

暂且不论成功与否，先看概念：这里的“I love you”就是 `明文` ，"J mpwf zpv"就是 `密文` ，向后顺延1个字母是 `加密` 过程，向前是 `解密` 过程，而这个规则就是 `算法` 。这种简单的加解密过程，就叫“对称加密”。缺点很显然，必须得打电话告诉女朋友怎么解密，岂不知隔墙有耳。

当然，更安全的方式是不要打电话也能处理。自然就是这里的私钥和公钥，它们都是长长的字符串值，私钥好比银行卡密码，公钥好比银行卡账户，账户谁都可以知道，但只有掌握私钥密码的人才能操作。不过，私钥和公钥更为贴心与先进，用私钥签名的信息，公钥可以认证确认，相反也可以。这就为网络传输和加密提供了便利。这就是“非对称加密”。

(2)加密货币地址

拿加密货币的鼻祖，比特币而言，一个比特币地址就是一个公钥，在交易中，比特币地址通常以收款人出现。如果把比特币交易必作一张支票，比特币地址就是收款人，也就是我们要写上收款人一栏的内容。

而私钥就是一个随机选出的数字而已，在比特币交易中，私钥用于生成支付比特币所必需的签名以证明资金的所有权，即地址中的所有资金的控制取决于相应私钥的所有权和控制权。

私钥必须始终保持机密，因为一旦被泄露给第三方，相当于该私钥保护之下的比特币也拱手相让了。私钥还必须进行备份，以防意外丢失，因为私钥一旦丢失就难以复原，其所保护的比特币也将永远丢失。

`Ebookcoin` 也是如此，只不过更加直接的把生成的公钥地址作为用户的ID，用作网络中的身份证明。更加强调用户应该仔细保存最初设定的长长的密码串，代替单纯的私钥保存，更加灵活。

(3)加密过程

Node.js的 `Crypto` 模块，提供了一种封装安全凭证的方式，用于HTTPS网络或HTTP连接，也对OpenSSL的Hash，HMAC，加密，解密、签名和验证方法进行了封装。

在币圈里，谈到加密技术时，经常听到Hash算法。很多小盆友时常与数组（array）和散列（hash）等数据格式混淆，以为Hash算法获得的结果都像json格式的键值对似的。

其实，这是语言上的差异， `Hash` 还有 `n. 混杂，拼凑; vt. 搞糟，把…弄乱` 的意思。所以，所谓的hash算法，解释为混杂算法或弄乱算法，更加直观些。

`Ebookcoin` 使用的是 `sha256` Hash算法（除此之外，还有MD5,sha1,sha512等），这是经过很多人验证的有效安全的算法之一（请看参考）。通过 `Crypto` 模块，简单加密生成一个哈希值：

```
var hash = crypto.createHash('sha256').update(data).digest()
```

这个语句拆开来看，就是`crypto.createHash('sha256')`先用`sha256`算法创建一个Hash实例；接着使用`.update(data)`接受`明文`data数据，最后调用`.digest()`方法，获得加密字符串，即`密文`。

然后，使用`Ed25519`组件，简单直接地生成对应密钥对：

```
var keypair = ed.MakeKeypair(hash);
```

(4)验证过程

加密技术的作用，重在传输和验证。所以，加密货币并不需要研究如何解密原文。而是，如何安全、快捷的验证。`Ebookcoin`使用了`Ed25519`第三方组件。

该组件是一个数字签名算法。签名过程不依赖随机数生成器，没有时间通道攻击的问题，签名和公钥都很小。签名和验证的性能都极高，一个4核2.4GHz 的 Westmere cpu，每秒可以验证 71000 个签名，安全性极高，等价于RSA约3000bit。一行代码足矣：

```
var res = ed.Verify(hash, signatureBuffer || ' ', publicKeyBuffer || ' ');
```

实践

在`Ebookcoin`世界里，`Ebookcoin`把用户设定的密码生成私钥和公钥，再将公钥经过16进制字符串转换产生帐号ID（类似于比特币地址）。付款的时候，只要输入这个帐号ID（或用户别名）就是了。该ID，长度通常是160比特（20字节），加上末尾的`L`后缀，也就是21字节长度。

因此，在使用的过程中会发现，软件（钱包程序）仅仅要求输入密码（通常很长），而不像传统的网站，还要用户名之类的信息。这通常就是加密货币的好处，即保证了安全，也实现了匿名。

`Ebookcoin`要求用户保存好最初设定的长长的明文密码串，它是找回帐号（保存着用户的加密货币财富）的真正钥匙。这比直接保管私钥方便得多，当然，风险也会存在，特别是那些喜欢用短密码的人。为此，`Ebookcoin`提供了二次签名（类似于支付密码）、多重签名等措施，弥补这些问题。

这里，仅研究一下用户ID的生成，体验上述过程，请看代码：

```
// modules/accounts 628行
shared.generatePublickey = function (req, cb) {
    var body = req.body;
    library.scheme.validate(body, {
        ...
        required: ["secret"]
```

```
    }, function (err) {
        ...
    // 644行
        privated.openAccount(body.secret, function (err, account) {
            ...
            cb(err, {
                publicKey: publicKey
            });
        });
    });
};

// 447行
privated.openAccount = function (secret, cb) {
    var hash = crypto.createHash('sha256').update(secret, 'utf8').digest();
    var keypair = ed.MakeKeypair(hash);

    self.setAccountAndGet({publicKey: keypair.publicKey.toString('hex')}, cb);
};

// 482行
Accounts.prototype.setAccountAndGet = function (data, cb) {
    var address = data.address || null;
    if (address === null) {
        if (data.publicKey) {
      // 486行
            address = self.generateAddressByPublicKey(data.publicKey);
          ...
        }
    }
    ...
  // 494行
    library.logic.account.set(address, data, function (err) {
        ...
    });
};

// modules/accounts 455行
Accounts.prototype.generateAddressByPublicKey = function (publicKey) {
    var publicKeyHash = crypto.createHash('sha256').update(publicKey, 'hex').digest();
    var temp = new Buffer(8);
    for (var i = 0; i < 8; i++) {
        temp[i] = publicKeyHash[7 - i];
    }

    var address = bignum.fromBuffer(temp).toString() + 'L';
    if (!address) {
        throw Error("wrong publicKey " + publicKey);
    }
    return address;
};
```

说明：上面628行，是产生公钥的方法，通常需要用户提供一个密码 `secret` 。447行，可以看到，将用户密码进行加密处理，然后直接生成了密钥对，接着将公钥继续处理。486行调用了方法 `generateAddressByPublicKey` ，455行，该方法对公钥再一次加密，然后做16进制处理，得到所要地址。

过程中，对于私钥没有任何处理，直接无视了。这是因为，这里的使用方法 `ed25519` ，基于某个明文密码的处理结果不是随机的，用户只要保护好自己的明文密码字符串，就可以再次生成对应私钥和公钥。

总结

加解密技术专业性很强，需要花费时间深入研究。我在前人研究成果的基础上进行了汇总和再加工，绘制了三张脑图，建议阅读，详情见第四部分《三张图让你全面掌握加密解密技术》一章。本篇权当入门，并没有对交易、区块链和委托人等的加密验证处理过程进行分析，加密解密过程都比较类似，后续阅读时会进一步说明。

其实，加密和验证的过程贯穿于亿书的全过程，接下来我们逐一揭开他们神秘的面纱。请看下一篇：地址

参考

Ed25519官方网站

地址

前言

上一篇，我们专门研究了《加密和验证》技术，明白了亿书对于加密解密技术的简单使用。我们说，加密货币处处都要用到加密解密技术，绝对不是一句空话，本篇文章我们就从最基础的加密货币地址开始，来学习和体会这句话的意义。

源码

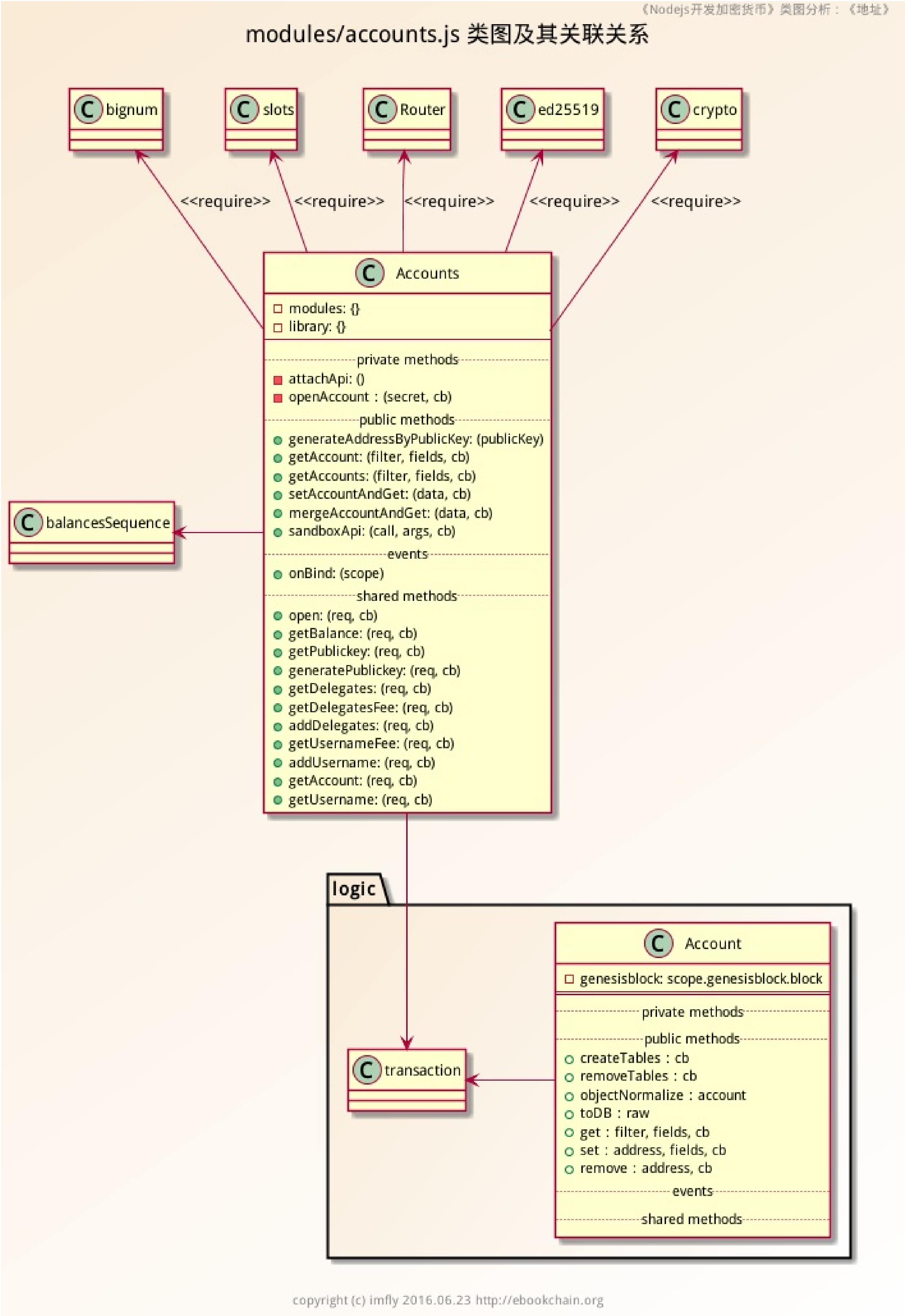
account.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/logic/account.js

accounts.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/accounts.js

contacts.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/contacts.js

类图

地址主要通过 `modules/accounts.js` 模块处理，类图如下：



《Nodejs开发加密货币》类图分析：《地址》
modules/accounts.js 类图及其关联关系
bignum
slots
Router
ed25519
crypto
<<require>>
<<require>>
<<require>>
<<require>>
<<require>>
Accounts
modules: {}
library: {}
private methods
attachApi: ()
openAccount : (secret, cb)
public methods
generateAddressByPublicKey: (publicKey)
getAccount: (filter, fields, cb)
getAccounts: (filter, fields, cb)
setAccountAndGet: (data, cb)
mergeAccountAndGet: (data, cb)
sandboxApi: (call, args, cb)
events
onBind: (scope)
shared methods
open: (req, cb)
getBalance: (req, cb)
getPublickey: (req, cb)
generatePublickey: (req, cb)
getDelegates: (req, cb)
getDelegatesFee: (req, cb)
addDelegates: (req, cb)
getUsernameFee: (req, cb)
addUsername: (req, cb)
getAccount: (req, cb)
getUsername: (req, cb)
balancesSequence
logic
Account
genesisblock: scope.genesisblock.block
private methods
public methods
createTables : cb
removeTables : cb
objectNormalize : account
toDB : raw
get : filter, fields, cb
set : address, fields, cb
remove : address, cb
events
shared methods
transaction
copyright (c) imfly 2016.06.23 http://ebookchain.org

流程图

这里的逻辑并不复杂，其本质就是一种`交易`，所以，我们将在《交易》那篇提供详细的流程图。

解读

计算机软件是人类活动的模拟和程序化，也就是大家所谓的“虚拟化”。在设计的时候，都会设想一个角色（Role）代替人类，负责完成要开发的各类操作。这个角色，通常在开发中被定义为用户（User），用户看到并可操作的就是用户帐号，在此基础上，才能进行权限认证，记录和管理与用户有关的各类操作。

比特币里的用户角色仅仅就是一个比特币地址，该地址是通过Hash算法进行加密处理的字符串，因此我们叫它`Hash地址`。同时，基于真实网络的复杂性，对于IP地址的追踪也不容易，所以比特币的匿名性很好（因为压根就没有给你暴露名字的机会）。

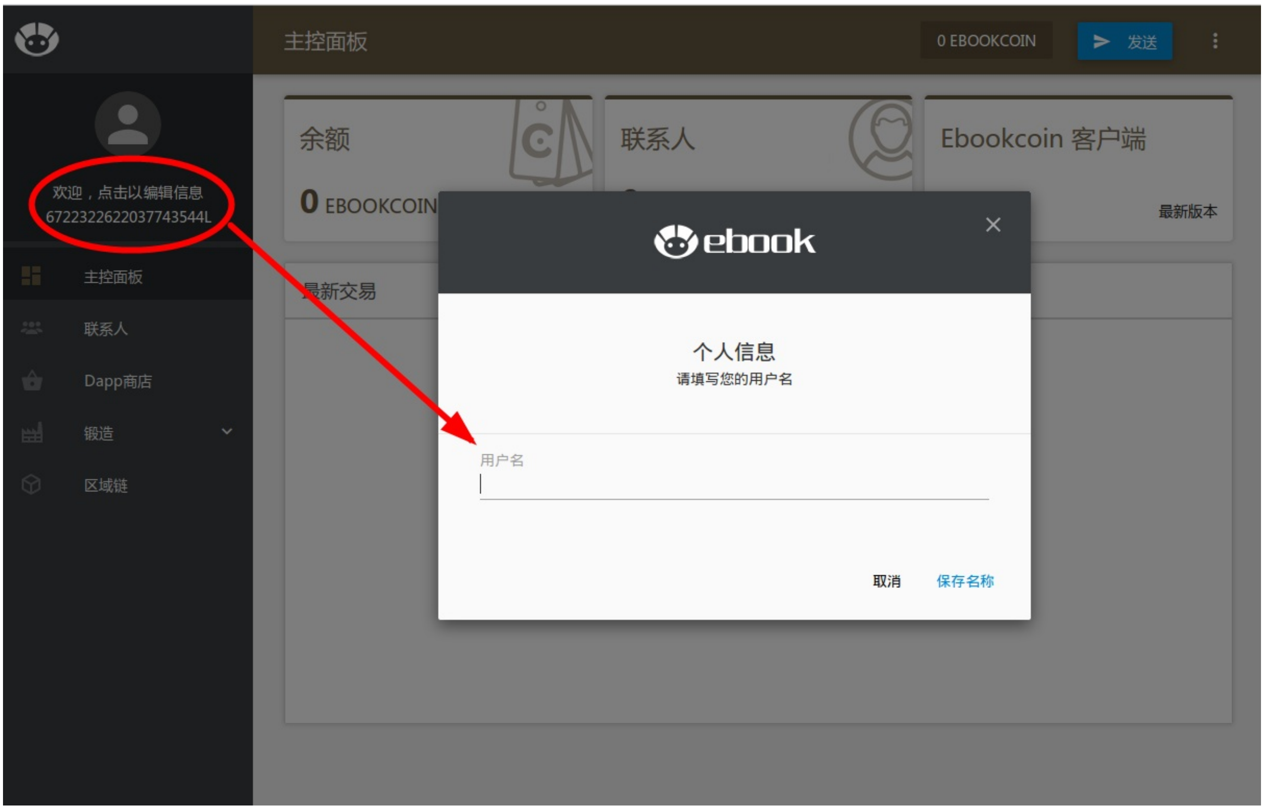
亿书作为一款加密货币产品，自然也提供了类似的`Hash地址`。并基于该地址，扩展提供了其他功能，比如“别名地址”。原因有三个：

1. 本质需要。版权保护应用，必然要明确版权所有人，实名信息是基本要求，如果再进行完全的匿名操作，显然有点不合适，属于跟自己过不去。当然，普通阅读用户不需要实名，亿书允许保持足够的匿名性。
2. 用户需要。复杂的字符串地址不适合人类脑记，很多人在最初接触比特币的时候，非常不习惯，经常弄混、忘记自己的比特币地址就是很好的证明。
3. 产品需要。说到交互功能，比特币除了交易之外，是没有什么交互的。而作为面向普通用户的亿书，要提供基本写作、团队协作、自出版等极具个性化的功能，交互功能被摆在突出位置，充满个性化的用户名是必须的。

具体操作时，可以实现下面的需求，详见 亿书白皮书 ：

1. 用户可以注册一个用户名，它相当于是用户帐户的一个别名;
2. 用户名都是唯一的;
3. 注册后无法更改或删除;
4. 用户名可作为支付地址，所以称为`别名地址`，类似于人们常用的支付宝帐号，其它用户可以直接向该用户的用户名付款，用户不再需要记下一长串的加密货币地址;
5. 用户可以维护一个联系人列表。

这些操作和信息，都可以在客户端里完成，如下图所示：



1.公共Api

看 `modules/accounts.js` 368行的代码， 如下：

```
// 368行
router.map(shared, {
  "post /open": "open",
  "get /getBalance": "getBalance",
  "get /getPublicKey": "getPublickey",
  "post /generatePublicKey": "generatePublickey",
  "get /delegates": "getDelegates",
  "get /delegates/fee": "getDelegatesFee",
  "put /delegates": "addDelegates",
  "get /username/get": "getUsername",
  "get /username/fee": "getUsernameFee",
  "put /username": "addUsername",
  "get /": "getAccount"
});

// 439行
library.network.app.use('/api/accounts', router);
```

前面，我们分析过，这里的router是 `helpers/router.js` 的一个实例。上述代码，最终会在439行的调用中，映射为公共Api，并分别对应 `shared` 中的方法，如：

```
// accounts
get /api/accounts/ -> shared.getAccount //帐号主页

post /api/accounts/open -> shared.open //登录
get /api/accounts/getBalance -> shared.getBalance
get /api/accounts/getPublicKey -> shared.getPublickey
post /api/accounts/generatePublicKey -> shared.generatePublickey

// username
get /api/accounts/username/get -> shared.getUsername
get /api/accounts/username/fee -> shared.getUsernameFee
put /api/accounts/username -> shared.addUsername //注册用户名

// delegates
get /api/accounts/delegates -> shared.getDelegates
get /api/accounts/delegates/fee -> shared.getDelegatesFee
put /api/accounts/delegates -> shared.addDelegates

// count 对应431行单独定义
get /api/accounts/count -> privated.accounts

// 另外两个是在debug或top环境下调试用的，暂且不表。
...
```

这里的 `delegates` （受托人）api，以及后面的debug环境下的api，暂且不提。通盘浏览这些公开接口信息，我们知道，可以直接浏览的信息主要包括余额（balance)、公钥（publicKey)、用户名（username）及修改用户名需要花费的费用（fee），以及受托人及其费用等，可以产生公钥和添加用户名。但是，没有删除和修改用户名的功能，所以一旦注册了用户名，想要修改，只能重新注册一个。

2.Hash地址

比特币地址是使用前缀来区分的，比如：1开头的地址就是我们实际使用的地址，3开头的地址是测试地址。而亿书，使用后缀来区分，通常以L结尾。代码在modules/accounts.js文件里：

```
// 455行
Accounts.prototype.generateAddressByPublicKey = function (publicKey) {
    var publicKeyHash = crypto.createHash('sha256').update(publicKey, 'hex').digest();
    var temp = new Buffer(8);
    for (var i = 0; i < 8; i++) {
        temp[i] = publicKeyHash[7 - i];
    }

    var address = bignum.fromBuffer(temp).toString() + 'L';
    if (!address) {
        throw Error("wrong publicKey " + publicKey);
    }
    return address;
};
```

另一个类似的地址，就是区块链的以C结尾的块地址，用来标注 `generatorId` ，代码与上面的基本一致：

```
// logic/block.js 20行
privated.getAddressByPublicKey = function (publicKey) {
    var publicKeyHash = crypto.createHash('sha256').update(publicKey, 'hex').digest();
    var temp = new Buffer(8);
    for (var i = 0; i < 8; i++) {
        temp[i] = publicKeyHash[7 - i];
    }

    var address = bignum.fromBuffer(temp).toString() + "C";
    return address;
}

// 上面的方法，在312行dbRead函数里调用，
generatorId: privated.getAddressByPublicKey(raw.b_generatorPublicKey),
```

3.别名地址

用户名相当于地址别名，就像支付宝帐号，所以白皮书说“别名地址”。我们知道，只有在转移支付的时候，才会用到地址（接受地址和发送地址），所以体现把用户名当作地址的逻辑代码，自然要在处理“资金转移”的交易代码里，看看 `modules/transactions.js` 文件，具体如下：

```
// modules/transactions.js 文件
// 652行
shared.addTransactions = function (req, cb) {
    var body = req.body;
    library.scheme.validate(body, {
        ...
        },
    // 685行
        required: ["secret", "amount", "recipientId"]
    }, function (err) {
    ...

    // 702行
        var isAddress = /^[0-9]+[L|l]$/g;
        if (isAddress.test(body.recipientId)) {
            query.address = body.recipientId;
        } else {
            query.username = body.recipientId;
        }

    library.balancesSequence.add(function (cb) {
            modules.accounts.getAccount(query, function (err, recipient) {
                ...
        // 717行
                var recipientId = recipient ? recipient.address : body.recipientId;
                var recipientUsername = recipient ? recipient.username : null;

                            ...

                            try {
                                var transaction = library.logic.transaction.create({
                  // 764行
                  type: TransactionTypes.SEND,
                                    amount: body.amount,
                                    sender: account,
                                    recipientId: recipientId,
                                    recipientUsername: recipientUsername,
                                    keypair: keypair,
                                    requester: keypair,
                                    secondKeypair: secondKeypair
                                });
                            } catch (e) {
                                return cb(e.toString());
                            }
                        ...
```

上述代码，实现的就是资金转账的功能（后台编码的交易类型 `TransactionTypes.SEND` ，见764行），毫无疑问，必须提供三个参数"secret", "amount", "recipientId"（见685行）。从702行可知，其中 `recipientId` 可以是字符串地址，也可以是用户名。从764行以后的代码还可以了解到，交易时的 `recipientId` 和 `recipientUsername` 字段都保存在数据库里了。

4.注册用户名

亿书默认不提供别名地址，需要用户注册。从上面的Api，很容易找到“注册用户名”的源码方法 `shared.addUsername` ，如下：

```
// 868行
shared.addUsername = function (req, cb) {
    var body = req.body;
    library.scheme.validate(body, {
        type: "object",
        properties: {
      ...
        },

    // 890行
        required: ['secret', 'username']
    }, function (err) {
      // 896行
        var hash = crypto.createHash('sha256').update(body.secret, 'utf8').digest();
        var keypair = ed.MakeKeypair(hash);

        if (body.publicKey) {
            if (keypair.publicKey.toString('hex') != body.publicKey) {
                return cb("Invalid passphrase");
            }
        }

        library.balancesSequence.add(function (cb) {
            if (body.multisigAccountPublicKey && body.multisigAccountPublicKey != keyp
air.publicKey.toString('hex')) {
                modules.accounts.getAccount({publicKey: body.multisigAccountPublicKey}
, function (err, account) {
          ...

                    modules.accounts.getAccount({publicKey: keypair.publicKey}, functi
on (err, requester) {
            ...
                        try {
              // 949行
                            var transaction = library.logic.transaction.create({
                                type: TransactionTypes.USERNAME,
                                username: body.username,
                                sender: account,
                                keypair: keypair,
                                secondKeypair: secondKeypair,
                                requester: keypair
                            });
                        } catch (e) {
                            return cb(e.toString());
                        }
                        modules.transactions.receiveTransactions([transaction], cb);
                    });
```

```
                });
            } else {
                self.getAccount({publicKey: keypair.publicKey.toString('hex')}, functi
on (err, account) {
                    ...

                    try {
            // 984行
                        var transaction = library.logic.transaction.create({
                            type: TransactionTypes.USERNAME,
                            username: body.username,
                            sender: account,
                            keypair: keypair,
                            secondKeypair: secondKeypair
                        });
                    } catch (e) {
                        return cb(e.toString());
                    }
                    modules.transactions.receiveTransactions([transaction], cb);
                });
            }

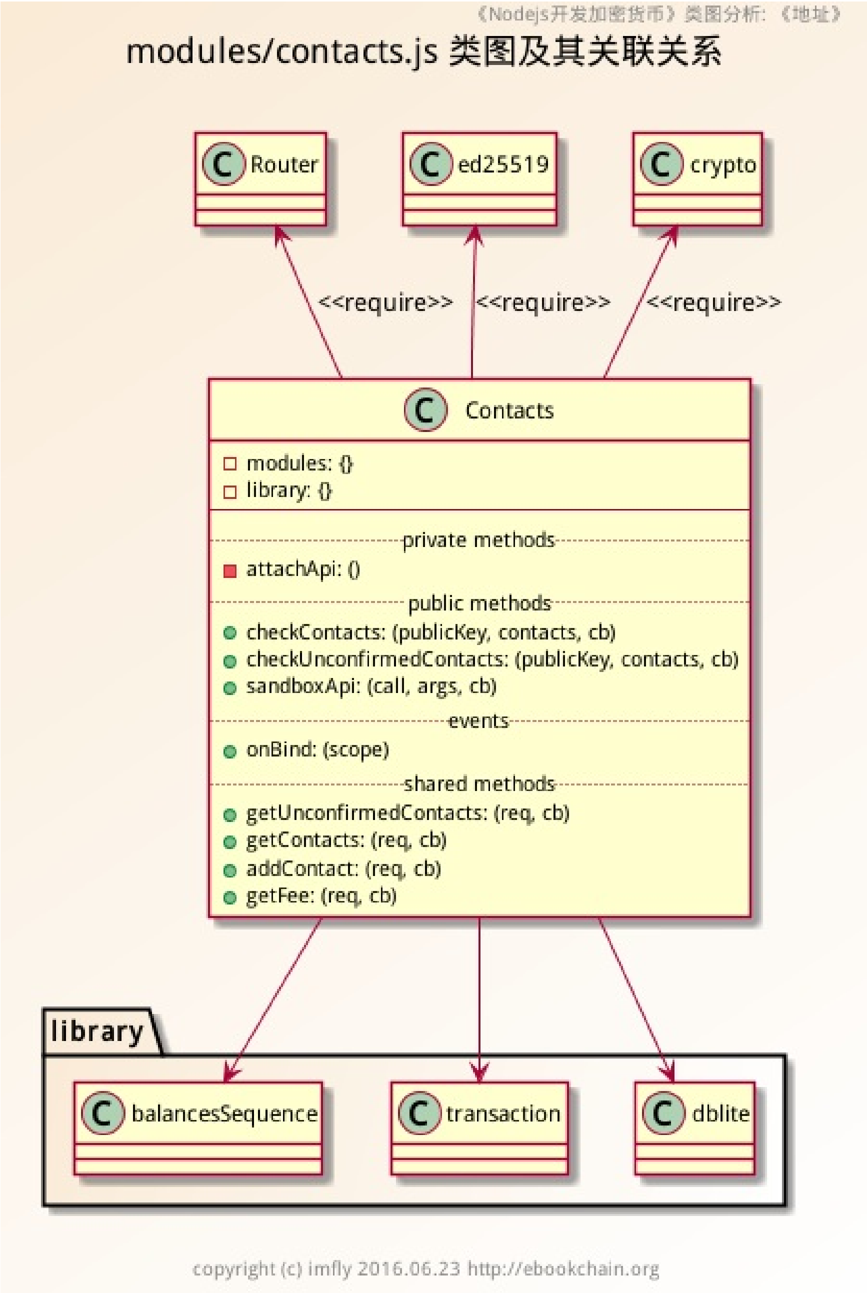
        ...
```

分析代码949行和984行，可以了解到，所谓的`注册用户名`，实质上就是提交了一个`TransactionTypes.USERNAME`类型的交易。890行代码，说明该Api必须两个参数，用户要提供明文密码(secret)和用户名(username)。

5.联系人列表

亿书具备社交功能，维护了一个联系人列表。与传统中心化软件不同的是，加密货币系统里，处处是交易，用户关注其他用户的行为，也是一项交易（内部的交易类型为`TransactionTypes.FOLLOW`）。

这项功能的源码在文件`modules/contacts.js`里，类图如下：



与其他模块文件一样，我们可以非常清晰的看到该文件提供的公共Api，如下：

```
// modules/contacts.js文件
// 198行
router.map(shared, {
  "get /unconfirmed": "getUnconfirmedContacts",
  "get /": "getContacts",
  "put /": "addContact",
  "get /fee": "getFee"
});
```

这些Api很简单，我们重点关注其中两个Api：

```
put /api/contacts -> shared.addContact //添加关注功能
get /api/contacts -> shared.getContacts //获得列表
```

对应方法的源码：

```
// 406行
shared.addContact = function (req, cb) {
    var body = req.body;
    library.scheme.validate(body, {
        ...
    // 431行
        required: ["secret", "following"]
    }, function (err) {
        ...

    // 448行
        var followingAddress = body.following.substring(1, body.following.length);
        var isAddress = /^[0-9]+[L|l]$/g;
        if (isAddress.test(followingAddress)) {
            query.address = followingAddress;
        } else {
            query.username = followingAddress;
        }

        library.balancesSequence.add(function (cb) {
            if (body.multisigAccountPublicKey && body.multisigAccountPublicKey != keyp
air.publicKey.toString('hex')) {
        ...
                        try {
                            var transaction = library.logic.transaction.create({
                                type: TransactionTypes.FOLLOW,
                                sender: account,
                                keypair: keypair,
                                secondKeypair: secondKeypair,
                                contactAddress: followingAddress, // 511行
                                requester: keypair
                            });
                        } catch (e) {
                            return cb(e.toString());
                        }
                        modules.transactions.receiveTransactions([transaction], cb);
                    });
                });
            ...
```

431行，添加关注需要两个参数"secret"和"following"，这里"following"其实就是用户的帐号地址或用户名（见448行）。然后，经过一系列验证之后，写入数据库的 `contactAddress` 字段（见511行），一个关注的操作过程就完成了。

然后，用户通过客户端浏览自己的联系人列表，需要用到另一个Api，对应的方法是`shared.getContacts`，如下：

```
shared.getContacts = function (req, cb) {
      // 362行
        modules.accounts.getAccount({address: query.address}, function (err, account)
{
            ...

            async.series({
                contacts: function (cb) {
          // 372行
                    if (!account.contacts.length) {
                        return cb(null, []);
                    }
                    modules.accounts.getAccounts({address: {$in: account.contacts}}, [
"address", "username"], cb);
                },
                followers: function (cb) {
                    if (!account.followers.length) {
                        return cb(null, []);
                    }
                    modules.accounts.getAccounts({address: {$in: account.followers}},
["address", "username"], cb);
                }
            ...
            });
        });
    });
};
```

这段代码最重要的部分，就是362行`modules.accounts.getAccount`方法的调用，获得对应地址的用户帐号（`account`）实例，联系人都保存在该实例的`account.contacts`里。

总结

读完代码，可以发现，代码逻辑非常简单，仅仅相当于两个基本功能，一个是生成加密货币的Hash地址，另一个是通过`交易`模块扩展和关联其他功能。其中，Hash地址是基础，在整个亿书的开发设计中，无处不在，签名、验证和交易，以及区块链等需要它。接下来我们介绍签名和验证，请看下一篇：签名和多重签名。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org

参考

亿书白皮书 http://ebookchain.org/ebookchain.pdf

签名和多重签名

前言

随着区块链等相关技术的创新和突破，很多有形或无形资产都将实现去中心化，数字资产将无处不在。比如我们这里分享的 亿书 就是要把数字出版物版权进行保护，实现去中心化，解决业界多年来版权保护不力的难题。

无论数字资产，还是数字出版版权，都是有明确所有权的，当前实现数字资产所属的技术手段就是本篇要介绍的 `签名` 。而 `多重签名` 是对 `签名` 的扩展使用，给数字资产转移提供了安全保障和技术手段。本篇，从基本概念入手，详细了解 `签名` 和 `多重签名` 的作用和代码实现。

源码

主要源码地址：

signatures.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/signatures.js

multisignatures.js
https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/multisignatures.js

解读

签名

（1）签名的作用和特点

名字的解释。 `签名` 是什么？有人第一反应是日常生活中的用笔 `签名` ，那么你的直觉是对的。不过，当很多小伙伴看到网上的解释，却又迷惑了，似乎与我们所知的 `签名` 相去甚远，事实上不同概念在不同领域的表述是有差别的，但本质相同。我坚信生活是一切创作的源泉，任何一个概念都能从生活中找到原型，这里的“签名”也是如此。

签名的作用。日常生活中，凡是需要确认归属的（是签名人的，不是其他的），都需要所有者进行签名。比如，我签名了一份文件，出了问题，责任我负，我签名了一个支票，就代表将由我支付。我们普通老百姓最常见的场景，就是去银行办业务，银行职员会让你反复签一大堆的单据，想必每个人都会有深刻的印象。

签名的特点。人的笔迹是很个性化的，越熟练的字体，个性特征越固定，因此一个人的名字，不同的人写出完全相同笔迹的概率非常小，即便是专业模仿也可以通过技术鉴别出来，这样一来，人的 `签名` 就具有唯一性、可验证的特点，并被法律认可。

签名的验证。如果，你拿着一张支票去银行兑换，银行职员会对支票上的签名和印章仔细比对，确保印章大小、样式，以及付款人签名等，与银行留存的信息一致，才会给你兑付，这就是 `签名` 验证。

（2）比特币客户端签名功能

数字资产需要签名。类比人类签名，比特币也有签名功能。如果了解比特币钱包（客户端软件），就会发现它提供了一个 `签名消息` 的功能，可以用来对其他用户通过比特币网络之外的信息进行签名和验证。我个人使用的是 比特币官方网站 提供的比太钱包，如图：



这个功能干什么用的呢？有好多小伙伴不清楚，这里举个简单的例子解释一下，具体使用的时候绝不限于这些应用。

Alice开了一个网店，但没有直接接入比特币网络，不能自动确认和验证支付者。客户Imfly购买了她的产品，并用比特币支付了全部货款。因为比特币地址和交易都是公开匿名的，为了防止冒充冒领，Alice需要确认Imfly提供的那个付款地址确实是imfly本人的，否则不能发货。这时候，就需要Imfly先把支付货款的比特币地址和相关交易 `签名信息` （如图），然后通过QQ或邮件传给Alice，Alice使用客户端 `验证信息签名` ，才能确认交易确实是Imfly的。

想象一下，如果没有`签名`功能会怎么样呢？因为比特币仅是一个匿名、安全的支付手段，但却无法确认支付方或收款方是谁，信息的不确定性，将使得比特币网络之外的交易无法达成。在中心化的世界里，这个问题是通过运营平台这个第三方达成的，比如支付宝等，双方的全部信息，平台都掌握，任何一方出现欺诈，都需要通过向平台投诉来解决。用户需要对第三方平台绝对信任，并通过牺牲个人信息安全获得交易的基本保障。

（3）电子签名

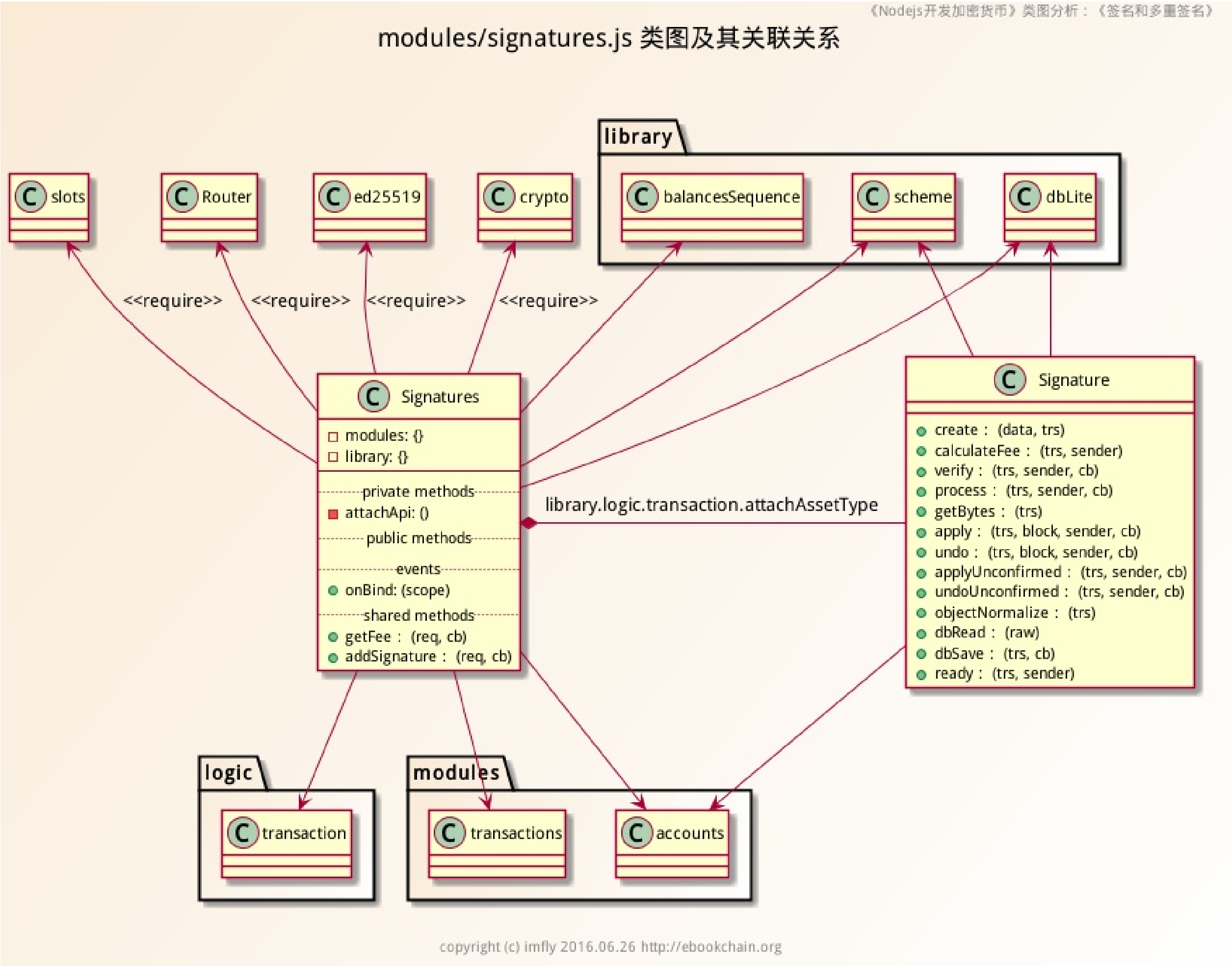
通过上述分析，可以理解的是，`签名`的作用是确定资产所属，其特征是简单、安全、可验证。把这个概念抽象出来，应用到计算机系统里，为了确定数字资产所属，也需要进行`签名`，这就是大家经常看到的“电子签名”的概念。在网络世界里，`签名`可以对任何需要确认的数字资产进行处理，比如比特币地址、电子书版权等，并以此来宣告重要资产的所属，这让无需监管的去中心化交易成为可能。

具体开发设计中，就是用加密技术代替人的笔迹，不然任何签名方法都会被模仿，而且模仿的成本极低，相反，验证的成本却很高。这也是当前数字出版行业版权保护不力的原因之一，各大平台仅仅通过用户权限来限制用户使用数字出版物，并没有对出版物本身采取数字签名等加密措施，一旦被盗版，验证和取证工作耗费很多人力物力。具体的加密或验证技术，请参考前面的章节。

亿书也具备签名能力，只不过，目前没有单独提供`签名信息`的操作供用户使用，而是通过签名，添加了`支付密码`功能，对用户帐号资产追加了一层保护。后续还会扩展更多的功能，对用户发布的任何数字出版物自动追加`电子签名`或`电子印章`，并保存到区块链。

（4）亿书的支付密码

`签名`方法在`modules/signatures.js`文件里，类图如下：



我们还是从Api开始，代码如下：

```
// modules/signatures.js文件
// 179行
router.map(shared, {
  "get /fee": "getFee",
  "put /": "addSignature"
});

// 188行
library.network.app.use('/api/signatures', router);
```

通过上面的代码，可以了解 `签名` 提供了两个简单的公共接口：

```
get /api/signatures/fee -> shared.getFee
put /api/signatures/ -> shared.addSignature //签名操作
```

显然，最核心的方法也就是 `shared.addSignature` ，代码：

```
// 215行
shared.addSignature = function (req, cb) {
    ...
    library.scheme.validate(body, {
        properties: {
            ...
        },
        required: ["secret", "secondSecret"]
    }, function (err) {
        ...

        library.balancesSequence.add(function (cb) {
            if (body.multisigAccountPublicKey && body.multisigAccountPublicKey != keyp
air.publicKey.toString('hex')) {
                modules.accounts.getAccount({publicKey: body.multisigAccountPublicKey}
, function (err, account) {
                    ...

                        try {
                            var transaction = library.logic.transaction.create({
                                type: TransactionTypes.SIGNATURE, // 297行
                                sender: account,
                                keypair: keypair,
                                requester: keypair,
                                secondKeypair: secondKeypair,

                            });
                        } catch (e) {
                            return cb(e.toString());
                        }
            ...
}
```

毫无疑问，`支付密码`也是一个简单的交易（交易类型`TransactionTypes.SIGNATURE`，见297行）。基于此，我们不难想象，添加类似比特币的`签名`功能也是件非常简单的事情，我们会在亿书下一个版本里添加这项功能，具体请关注 亿书币版本库 最新进展。

多重签名

上面我们提到，比特币的匿名性，使交易处于不可信之中，最终导致用户不敢交易。有了签名功能，就有了确认双方信息的有效手段，问题总算有了解决方案。聪明的小伙伴会发现，签名和验证过程除了繁琐，并没有让我们觉得比使用第三方平台更有效、更安全。有没有更好的解决方案呢？回答是：有，那就是多重签名。

（1）基本概念

多重签名，可以简单的理解为一个数字资产的多个签名。签名标定的是数字资产所属和权限，多重签名预示着数字资产可由多人支配和管理。在加密货币领域，如果要动用一个加密货币地址的资金，通常需要该地址的所有人使用他的私钥（由用户专属保护）进行签名。那么，多重签名，就是动用这笔资金需要多个私钥签名，通常这笔资金或数字资产会保存在一个多重签名的地址或帐号里（就比特币而言，多重签名地址通常以 `3` 开头）。这就好比，我们工作中有一份文件，需要多个部门签署才能生效一样。

在实际的操作过程中，一个多重签名地址可以关联n个私钥，在需要转账等操作时，只要其中的m个私钥签名就可以把资金转移了，其中m要小于等于n，也就是说m/n小于1，可以是2/3, 3/5等等，是要在建立这个多重签名地址的时候确定好的。

（2）工作原理

数字资产在某种情况下，需要多人支配。换句话说，在某些特定条件下，数字资产如果无法确认归属某个特定的人，那么最好让相关人共同签署它的所有权。

仍然举上面的例子，在Alice发货之后，Imfly收到货之前，这笔钱应该由第三方信用比较高的中介暂时保存，这个阶段，这笔钱要么是Alice的，要么是Imfly的，最终的归属要看Imfly是否收到货。所以，这个第三方，无论如何都是应该有的，不然Imfly就要承担大部分风险（因为比特币的单向不可逆，Imfly发送之后就没有办法收回了）

这样一来，这笔钱的所属关系，在交易过程中涉及到Alice、Imfly和平台第三方（虽然不属于它，但它有权裁定资金去向），那么就应该由他们三方签名，因此网上购物就是典型的多重签名的例子。其多重签名模型就是2/3，也就是说只要他们中的两个签名，资金就可以被转移。

具体到这个例子，Imfly把钱打给一个关联三方私钥的多重签名地址，如果整个交易过程顺利，只要Alice和Imfly两个签名，这笔钱就会顺利到达Alice手里。如果不顺利，他们任何一人提出仲裁，平台第三方调查之后，通过签名就能把这笔钱转给Alice或退回Imfly。这非常类似淘宝和京东的模式，但是比他们更加便捷和安全，至少不用担心第三方倒闭、挪用资金或携款跑路。

（3）应用场景

很显然，多重签名给了加密货币腾飞的翅膀，让它单一单项支付的能力更具吸引力，让加密货币技术应用到各行各业成为可能。这里简单的罗列几个应用场景，供探索和思考：

- 电子商务。比较常见的是2/3的模式。上面电子商务网站的例子，就是最典型的场景之一，目前已经有成功的案例了。延伸一下，这类应用本质就是中介，所以还可用在各类中介机构性质的服务上。

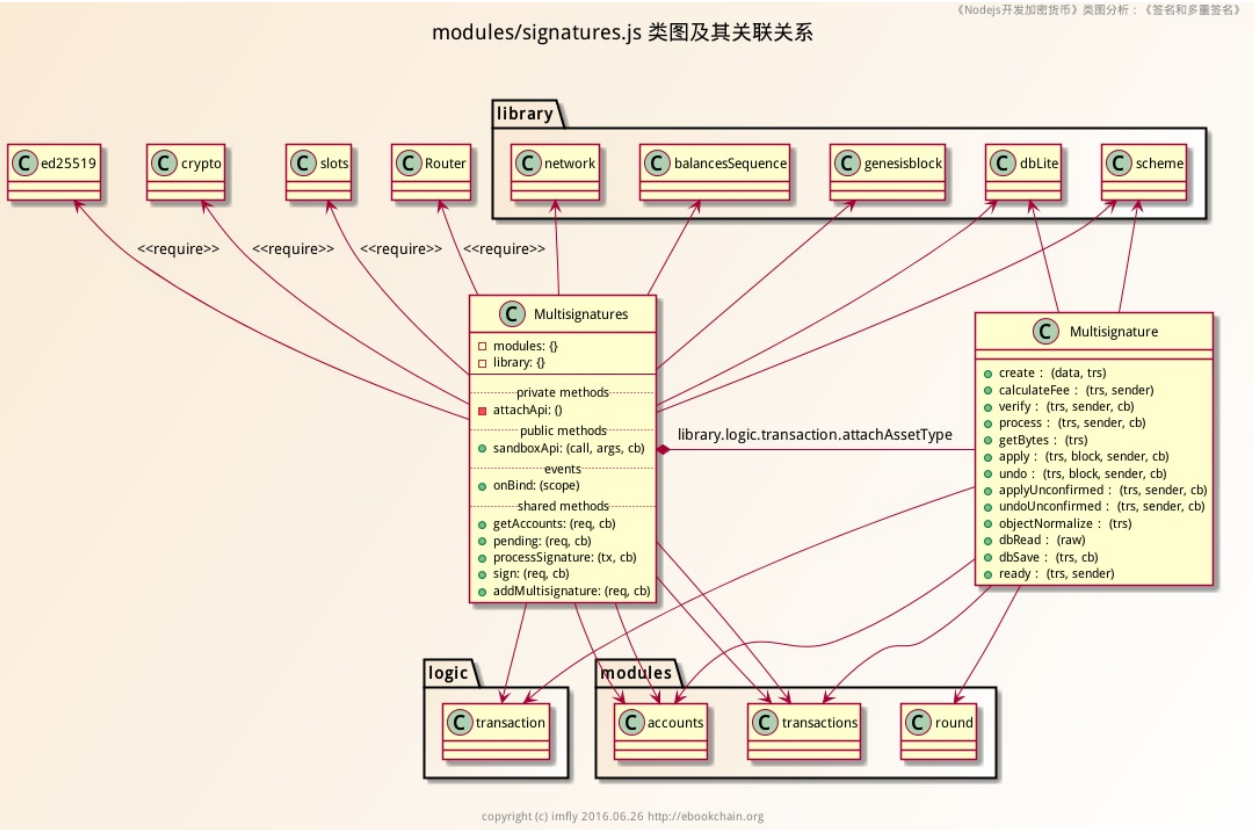
- 财产分割。比如夫妻双方共有财产，可以使用1/2的模式，一个账户谁都可以使用，跟各自拥有帐号一样，好处是系统忠实记录了每个人的花销，闹掰的时候很容易清算。扩展到公司合伙经营，可以使用1/n模式，n个人合伙人，都可以直接支配共有资金，具体清算时，一目了然。

- 资金监管。其实，这是多重签名的最直接作用，一笔钱需要多个人签名才能使用，任何一个人都无法直接动用资金，这在生活中太常见了，只要灵活设置多重签名的比重模式，就能解决生活中很多问题。比如，接着上面夫妻的例子，夫妻要储备一笔资金，供孩子上大学使用，在这之前谁都不能动，那么把模式改为2/2，不仅限制了夫妻双方，也给黑客攻击增加了难度。

多重签名的设计，让各种业务去中心化充满无限可能。

（4）亿书的多重签名

`多重签名` 方法在 `modules/multisignatures.js` 文件里，类图如下：



实现Api的代码如下：

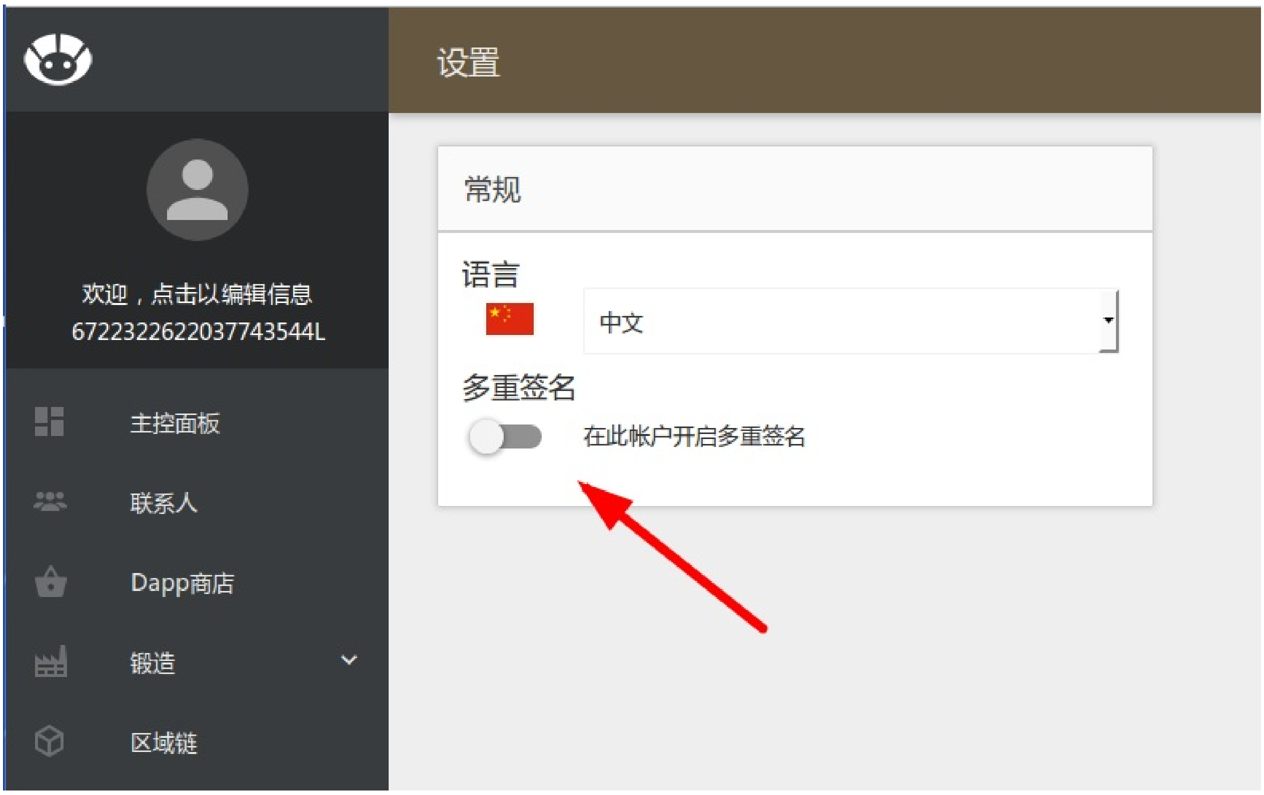
```
// 318行
router.map(shared, {
  "get /pending": "pending", // Get pending transactions
  "post /sign": "sign", // Sign transaction
  "put /": "addMultisignature", // Add multisignature
  "get /accounts": "getAccounts"
});

// 329行
library.network.app.use('/api/multisignatures', router);
```

解析一下，最后产生的Api如下：

```
get /api/multisignatures/pending -> shared.pending // 查询等待中的交易
post /api/multisignatures/sign -> shared.sign // 签名交易
put /api/multisignatures/ -> shared.addMultisignature // 创建多重签名帐号
get /api/multisignatures/accounts -> shared.getAccounts // 获得关联的帐号（对应者用户私钥）
```

提供的功能很显然，包括：待交易查询、关联帐号列表查询，用户签名交易，创建多重签名帐号等4个核心功能。我们先从创建多重签名帐号开始，这个Api使用的是http的 `put` 方法，对应的自然是 `更新` 操作，不查看代码也可以猜想到，该功能应该是在已有帐号基础上的操作，从客户端钱包 `设置` 菜单里，可以看到如图操作：



看看 `shared.addMultisignature` 的源代码如下：

```
// modules/multisignatures.js文件
shared.addMultisignature = function (req, cb) {
    var body = req.body;
    library.scheme.validate(body, {
        ...

    // 732行
        required: ['min', 'lifetime', 'keysgroup', 'secret']
    }, function (err) {
      ...

        library.balancesSequence.add(function (cb) {
            modules.accounts.getAccount({publicKey: keypair.publicKey.toString('hex')}
, function (err, account) {
                ...

        // 767行
                try {
                    var transaction = library.logic.transaction.create({
                        type: TransactionTypes.MULTI, // 769行
                        sender: account,
                        keypair: keypair,
                        secondKeypair: secondKeypair,
                        min: body.min,
                        keysgroup: body.keysgroup,
                        lifetime: body.lifetime
                    });
                } catch (e) {
                    return cb(e.toString());
                }

                ...
};
```

从732行可知，创建一个多重签名，必须'min', 'lifetime', 'keysgroup', 'secret'这四参数（其实，一个默认参数就是当前帐号），min代表上面讲到的 `m` 值，即需要确认的人数;lifetime代表生命周期;keysgroup包含多重签名关联的全部帐号，它是数组类型，包含的元素个数就是 `n` ，secret是用户密码，与用户私钥对应。

经过一系列的验证之后，作为一个交易（交易类型TransactionTypes.MULTI，769行）保存到数据库(区块链)里。创建成功的帐号，可以显示多重帐号菜单，对交易进行操作。接下来，自然应该能够查看全部关联的帐号（请看shared.getAccounts方法），查看 `待确认的交易` （请看shared.pending方法），这两个方法仅仅是简单的查询，没什么难度，这里不再浪费篇幅。

如果用户同意交易，就可以对 `待确认的交易` 进行签名（shared.sign方法），这个方法的源码如下：

```
// 586行
shared.sign = function (req, cb) {
    var body = req.body;
    library.scheme.validate(body, {
        ...
        required: ['transactionId', 'secret']
    }, function (err) {
        ...

    // 632行
        function done(cb) {
            library.balancesSequence.add(function (cb) {
        // 634行
                var transaction = modules.transactions.getUnconfirmedTransaction(body.
transactionId);

                if (!transaction) {
                    return cb("Transaction not found");
                }

        // 640行
                transaction.signatures = transaction.signatures || [];
                transaction.signatures.push(sign);

                library.bus.message('signature', {
                    signature: sign,
                    transaction: transaction.id
                }, true);
                cb();
            }, function (err) {
                if (err) {
                    return cb(err.toString());
                }

                cb(null, {transactionId: transaction.id});
            });
        }

        ...
};
```

这个方法，相比单独的签名方法，不同的是单独的签名方法相当于一个新建交易，而这里的多重签名的用户签名，显然仅仅是对未确认交易（634行）进行签名确认（640行维护了一个签名数组，641行的push方法把用户签名写入数组）。而且，相比独立签名，验证也更复杂，我们将在下一篇《交易》一文中集中讨论验证问题。您也可以结合下一篇的内容，阅读和理解这里的签名方法。

总结

在加密货币里，每一个交易都涉及到使用私钥签名，用于确认每笔资金所有人。确定了所有人，自然就确定了资金转移的条件、目标和方向，就为我们下一步进行资金转移操作奠定了基础。很自然的，该研究一下亿书的交易了，请看下一篇：《交易》。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org

参考

亿书白皮书 http://ebookchain.org/ebookchain.pdf

交易

前言

我们在第一部分《了解加密货币》里说过，加密货币是“利益”转移的程序化，其核心目标是保证数字财富或价值安全、透明、快速的转移。因此，交易是加密货币系统中最重要的部分，是加密货币的核心功能，加密解密、P2P网络、区块链等一系列技术都是围绕交易展开的。

这一篇，我们就来研究亿书提供的交易类型及代码实现，集中总结交易的生命周期及实现过程，把我们在《地址》和《签名和多重签名》里故意漏掉的判断逻辑补充完整。

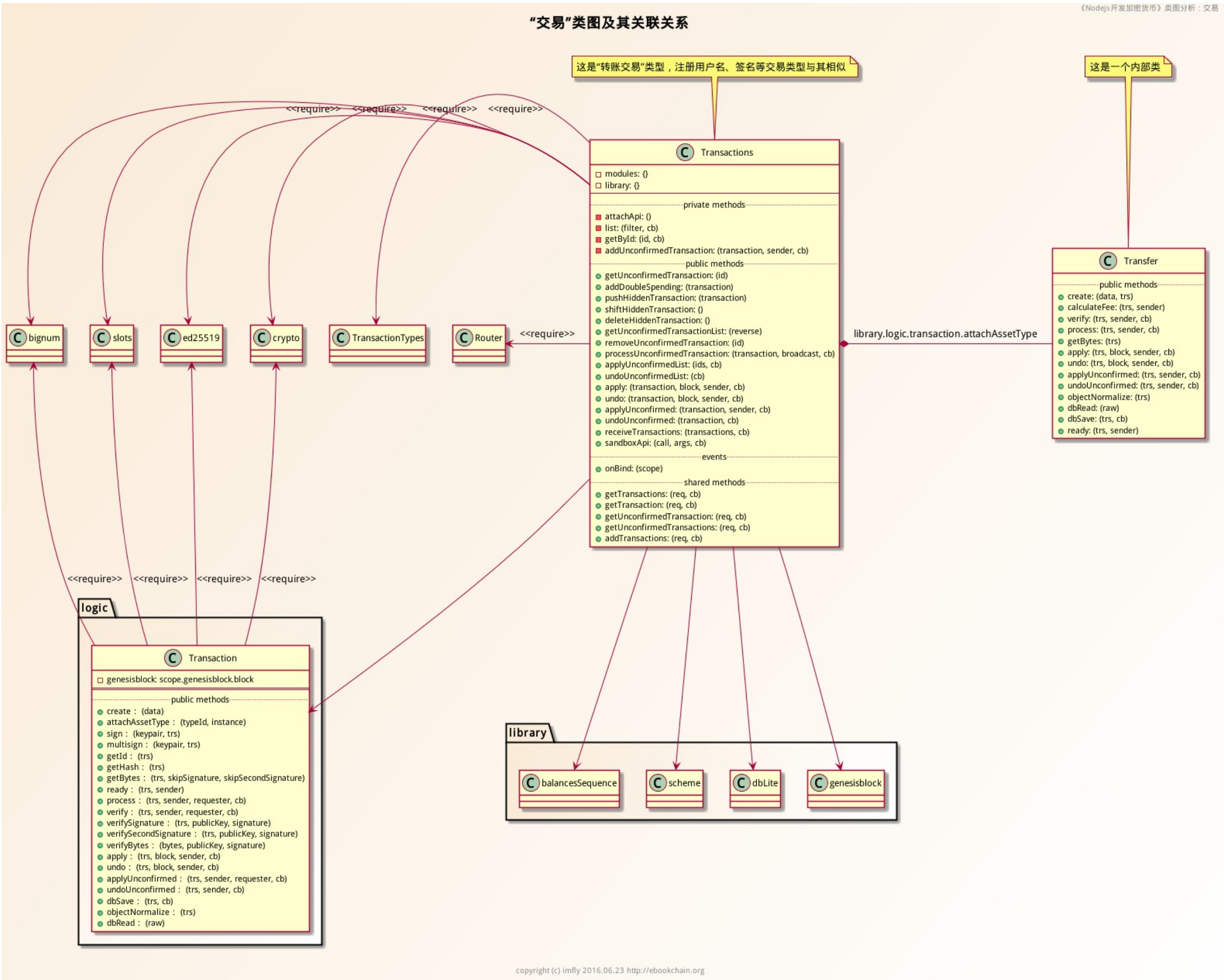
源码

transaction-types.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/helpers/transaction-types.js

transaction.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/logic/transaction.js

transactions.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/transactions.js

类图



解读

交易的本质

从经济学角度来说，交易就是一种价值交换。在《精通比特币》（见参考）一书里，作者是这样定义比特币交易的：简单地说，交易是指把比特币从一个地址转到另一个地址。更准确地说，一笔"交易"就是一个经过签名运算的，表达价值转移的数据结构。每一笔"交易"都经过比特币网络传输，由矿工节点收集并封包至区块中，永久保存在区块链某处。

交易，在汉语词典里，既可以是名词，代表交易内容的数据信息（技术上叫做数据结构），又可以是动词，代表一个操作过程。把这些重要信息汇总到一起，既让用户容易理解，又要体现加密货币特点，可以这样定义一个交易操作：

> 加密货币交易是指人们通过加密货币网络，把加密货币进行有效转移， 并把交易数据保存到区块链的过程。

这个定义与我们的直观感受比较接近。通常，大家喜欢把加密货币交易，比做纸质支票，支票本身就是记录一笔交易的数据结构，从签署支票到兑付完成的过程就是一个交易操作行为。一笔加密货币交易就是一个有着货币转移目的的电子支票，只有在交易被执行时才会在

金融体系中体现，而且交易发起人并不一定是签署该笔交易的人。

交易可以被任何人在线上或线下创建，即便创建这笔交易的人不是这个账户的授权签字人。这一点非常好理解，假如有一张空的纸质支票，我们可以自己填写，也可以找人填写，最后只要有支付权限的领导签名，支票就能生效，就可以兑付。加密货币也是如此，无论谁创建的加密货币交易，只要被资金所有者（们）数字签名，交易就能实现。

交易只是一些经过加密处理的字节码，不含任何机密信息、私钥或密码，可被包括wifi、无线电在内的任何网络公开传播，甚至可以被处理成二维码、表情符号、短信等形式发送。只要这笔交易能进入加密货币网络，那么发送者并不需要信任用来传播该笔交易的任何一个网络节点。同时，这些节点也不需要信任发送者，不用记录发送者的任何身份信息。相反，电子商务网站的交易，不仅包含敏感信息，而且依赖加密网络连接完成信息传输。

因此，从本质上讲，加密货币交易是价值所有权的变更，价值转移仅仅是这种行为的结果。加密货币总量就是那些，从始至终都不会变化，人为丢失的是人类流通使用的私钥权限，总量仍在网络上不会丢失。记录加密货币总量的区块链就那一条，这个链条可以越来越长，越来越大，但是增加的仅仅是交易信息，即价值所有权变更信息。用个不慎确切的比喻，加密货币就像一列永不停息的火车，上下的是人次，固定的是座位，您只有在自己的人生旅途中才拥有某个座位的所有权（使用权）。

从设计原理上说，加密货币淡化了交易者帐号，简化为输入输出，所谓的账户也只是存在于客户端钱包这类具体的应用层的软件里，就像那列火车总要有火车站吧，而某一段旅程的火车票是有具体所属的，是要与现实人的帐号或身份对应的，所以火车站是要记录用户信息，要有检票、验票的过程。

亿书的原理也是如此，只不过亿书通过进一步扩展交易类型，强化了用户帐号的存在，使得更加适合处理各类资产所有权，从而为数字版权保护奠定良好架构基础。

交易生命周期

加密货币的整个系统，都是为了确保正确地生成交易、快速地传播和验证交易，并最终写入全球交易总账簿——区块链而设计。因此，从开发设计角度考虑，一笔交易必须包括下列过程：

1. 生成一笔交易。这里是指一条包含交易双方加密货币地址、数量、时间戳和有效签名等信息，而且不含任何私密信息的合法交易数据;
2. 广播到网络。几乎每个节点都会获得这笔交易数据。
3. 验证交易合法性。生成交易的节点和其他节点都要验证，没有得到验证的交易，是不能进入加密货币网络的。
4. 写入区块链。

下面，我们来详细阅读分析亿书的交易是如何实现的。

亿书交易类型

目前，亿书已经完成或正在开发的交易类型，包括14种（后续会有更多），分别是：

```
// helpers/transaction-types.js
module.exports = {
    SEND : 0,
    SIGNATURE : 1,
    DELEGATE : 2,
    VOTE : 3,
    USERNAME : 4,
    FOLLOW : 5,
    MULTI: 6,
    DAPP: 7,
    IN_TRANSFER: 8,
    OUT_TRANSFER: 9,
    ARTICALE : 10,
    EBOOK: 11，
  BUY: 12，
    READ: 13
}
```

其中，

`SEND` 是最基本的转账交易， `SIGNATURE` 是上一篇提到的“签名”交易， `DELEGATE` 是注册为受托人， `VOTE` 是投票， `USERNAME` 是注册用户别名地址， `FOLLOW` 是添加联系人， `MULTI` 是注册多重签名帐号， `DAPP` 是侧链应用， `IN_TRANSFER` 是转入Dapp资金， `OUT_TRANSFER` 转出Dapp资金，这些是现有版本已经完成的功能。

`ARTICALE` 是发布文章， `EBOOK` 是发布电子书， `BUY` 是购买（电子书或其他商品）， `READ` 是付费阅读（电子书等），这些功能会逐步添加。

这些交易，除了 `SEND` 转账交易外，其他的交易类型，我们暂且称它们为功能性交易（在比特币的圈子里，有人称为伪交易）。

交易基本流程

亿书交易类型尽管多样，但是交易的基本逻辑是一样的。整个加密货币都是交易逻辑的有效组成部分，要比传统电子商务网站复杂的多，但与交易直接相关的代码，却又非常简单清晰。从开发角度说，实现一笔交易，亿书需要这样几个步骤：

（1）生成交易数据

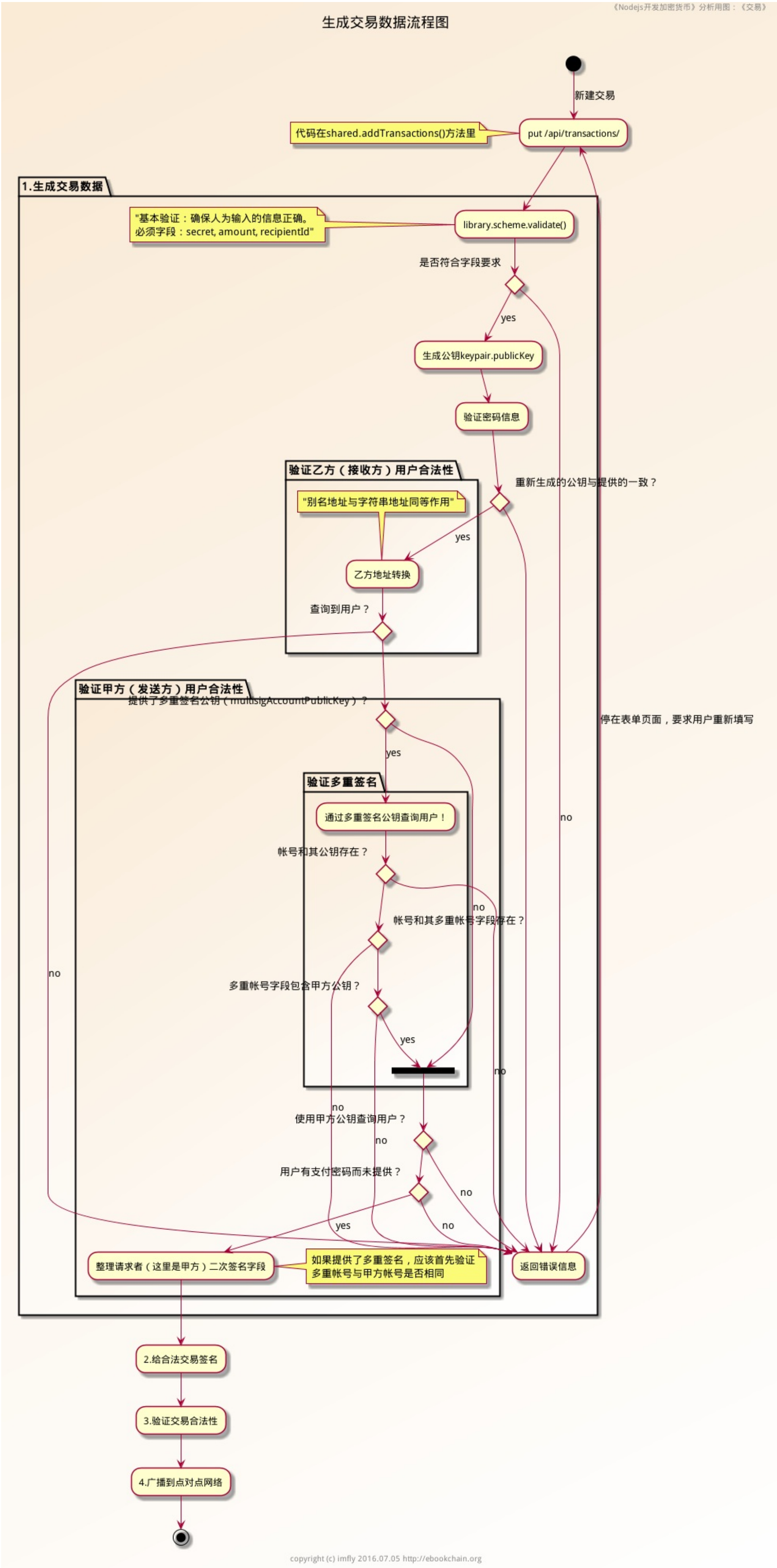
交易是人类行为，涉及到甲乙双方（货币发送者和接收者，我们用甲乙方来代替，下文同）和交易数额，这在很多交易，特别是版权交易方面更加重要。甲方是主动发起交易的有效用户，是亿书币的支付方，是交易的支付来源。乙方比较灵活，可以是另一个有合法地址的用户，也可以是亿书系统本身（功能性交易），是亿书币的接收方。

简单的一句话就是：谁与谁交易了多少钱。用下面转账交易部分的代码举例，请看 `modules/transactions.js` 文件里的763和800行，一笔交易必须包含如下字段：

- 交易类型。代码里表示为 type: TransactionTypes.SEND;
- 支付帐号。代码里指的是 sender: account;
- 接受帐号。代码里指的是 recipientId: recipientId, 如果用的是别名地址，就是 recipientUsername: recipientUsername，如果是功能性交易，这里就不需要了;
- 交易数量。代码里指的是 amount: body.amount。

这些数据有的要求用户输入，比如用户密钥，交易数量等，这些数据是否正确，也是非常关键的事情。这是软件程序验证逻辑的一个重要部分，不可或缺。这个很好理解，如果一个人胡乱填写密钥和接受地址，也能把币发送出去，那就笑话了。但具体校验过程较为繁琐，这里主要涉及到：发起交易的用户是否存在、密钥是否正确、是否多重签名帐号、是否有支付密码，以及接受方用户地址是否合法等，都要逐个检验。

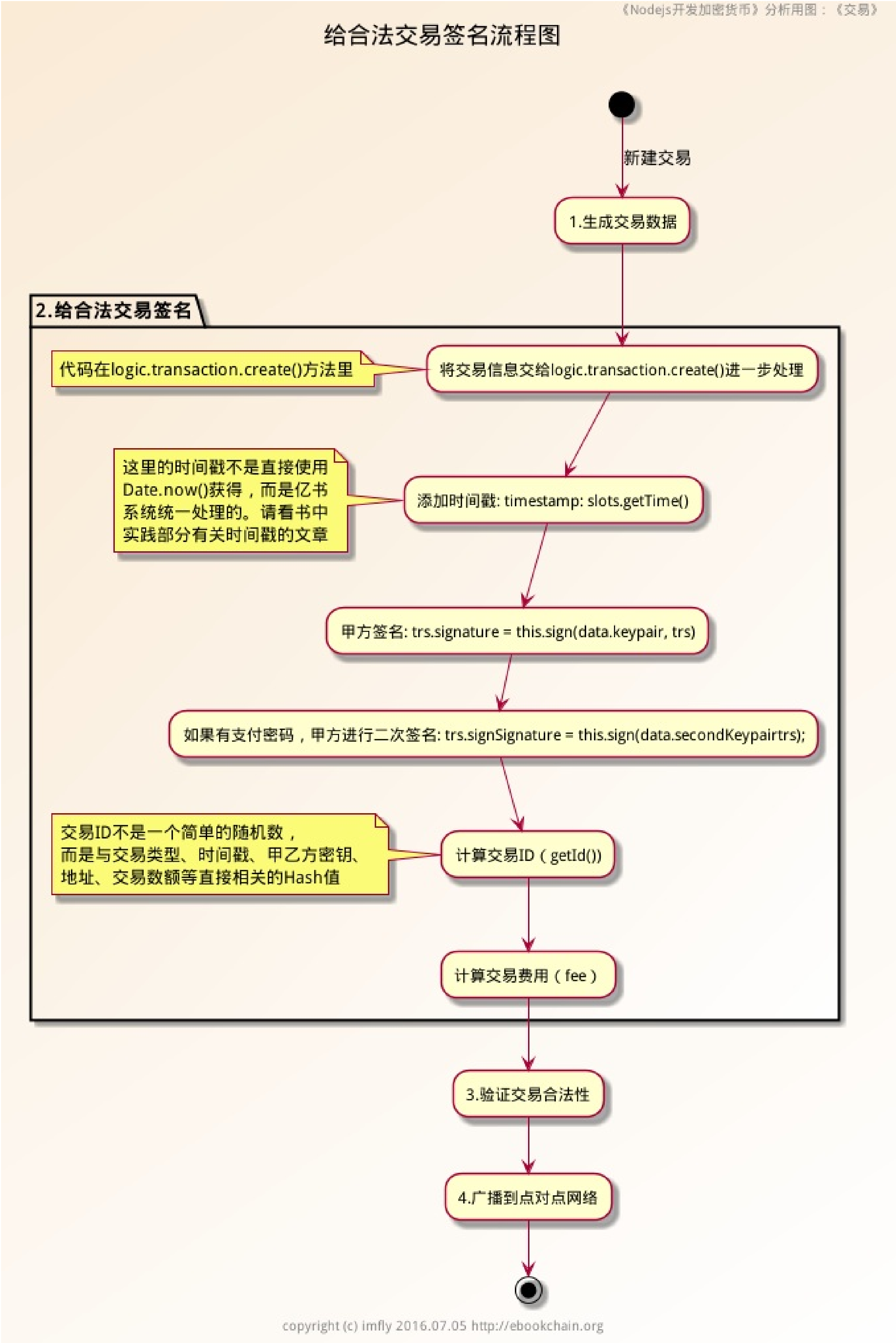
详情看这里的流程图：


《Nodejs开发加密货币》分析用图：《交易》
生成交易数据流程图
新建交易
代码在shared.addTransactions()方法里
put /api/transactions/
1.生成交易数据
"基本验证：确保人为输入的信息正确。
必须字段：secret, amount, recipientId"
library.scheme.validate()
是否符合字段要求
yes
生成公钥keypair.publicKey
验证密码信息
验证乙方（接收方）用户合法性
重新生成的公钥与提供的一致？
"别名地址与字符串地址同等作用"
yes
乙方地址转换
查询到用户？
验证甲方（发送方）用户合法性
提供了多重签名公钥（multisigAccountPublicKey）？
停在表单页面，要求用户重新填写
yes
验证多重签名
通过多重签名公钥查询用户！
no
帐号和其公钥存在？
no
帐号和其多重帐号字段存在？
no
多重帐号字段包含甲方公钥？
yes
no
no
使用甲方公钥查询用户？
no
用户有支付密码而未提供？
no
yes
no
整理请求者（这里是甲方）二次签名字段
如果提供了多重签名，应该首先验证
多重帐号与甲方帐号是否相同
返回错误信息
2.给合法交易签名
3.验证交易合法性
4.广播到点对点网络
copyright (c) imfly 2016.07.05 http://ebookchain.org

（2）给合法交易签名

基本信息正确之后，一笔合法交易，还要使用甲乙方的公钥签名，确保交易所属。同时，还要准确记录它的交易时间戳，方便追溯。还要生成交易ID，每个交易ID都包含了丰富的加密信息，需要复杂的生成过程，绝不像传统的网站系统，让数据库自动生成索引就可以充当ID了。

详情看这里的流程图：


《Nodejs开发加密货币》分析用图：《交易》
给合法交易签名流程图
新建交易
1.生成交易数据
2.给合法交易签名
代码在logic.transaction.create()方法里
将交易信息交给logic.transaction.create()进一步处理
这里的时间戳不是直接使用Date.now()获得，而是亿书系统统一处理的。请看书中实践部分有关时间戳的文章
添加时间戳: timestamp: slots.getTime()
甲方签名: trs.signature = this.sign(data.keypair, trs)
如果有支付密码，甲方进行二次签名: trs.signSignature = this.sign(data.secondKeypairtrs);
交易ID不是一个简单的随机数，而是与交易类型、时间戳、甲乙方密钥、地址、交易数额等直接相关的Hash值
计算交易ID（getId()）
计算交易费用（fee）
3.验证交易合法性
4.广播到点对点网络
copyright (c) imfly 2016.07.05 http://ebookchain.org

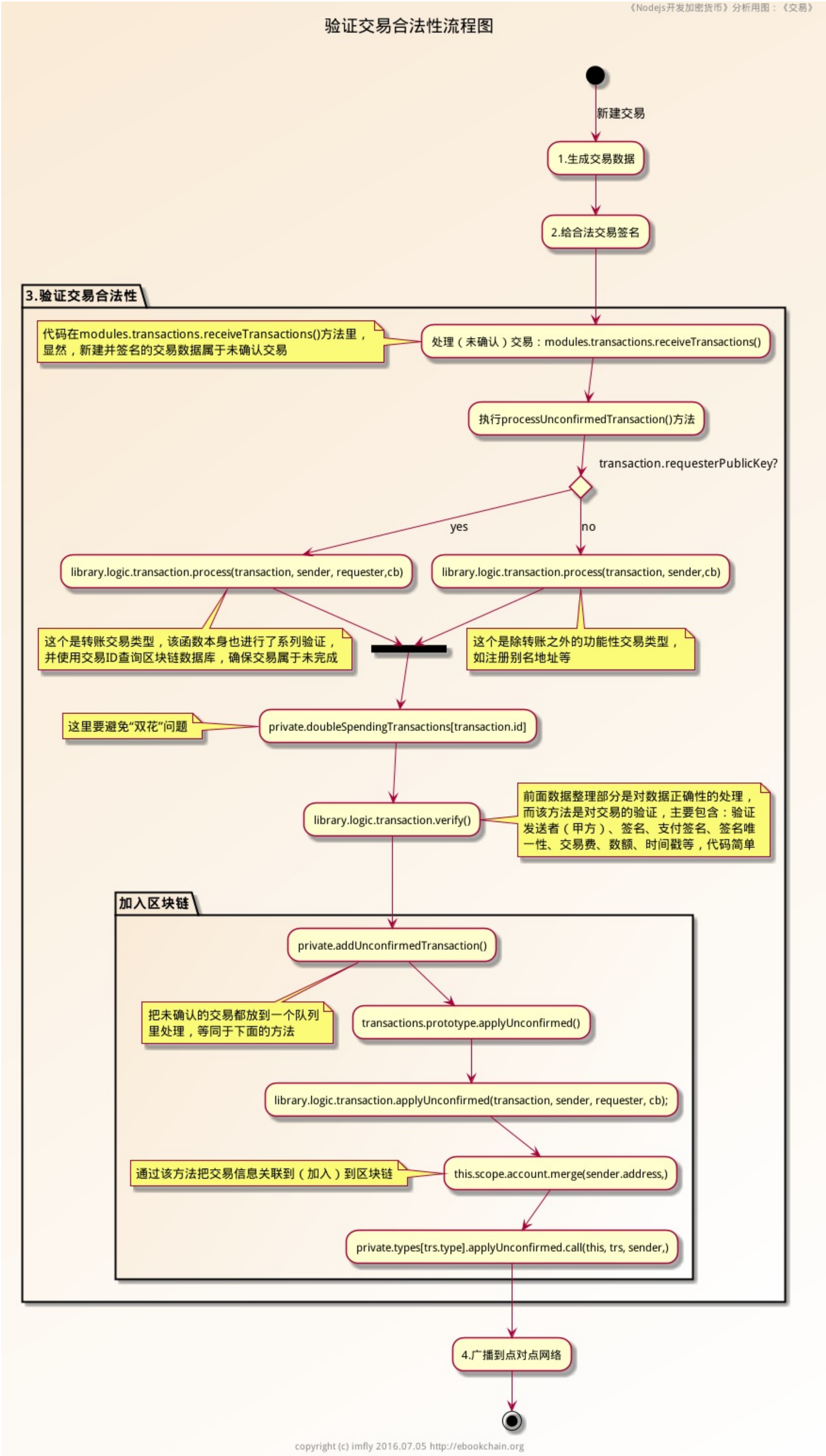
（3）验证交易合法性

通常，一笔交易经过6-10个区块之后，这笔交易被认为是无法更改的，即已确认，因为这时候拒绝、变更的难度已经非常大，理论上已经不可能。这里的交易合法性，除了基本信息正确之外，主要是指保证交易是未确认的交易，也不是用户重复提交的交易，即双花交易。双花交易是加密货币特有的现象，通俗的说，就是用户在交易确认之前（有一段时间，比特币时间更长），又一次提交了相同交易信息，导致一笔钱花两次，这种情况是必须要避免的。

每笔交易在广播到网络之前必须验证合法性，不合法的交易没有机会广播到网络。节点收到新的交易信息时，要重新验证。如此一来，任何对网络的攻击，都只会影响一个节点，安全性大大提高。

验证合法的交易就可以直接加入区块链了，因此从上面的第一步到现在，亿书都是在一个节点上完成的。这也为下面的广播处理打下基础，一旦交易被广播到网络，在其他节点，这里的验证和处理过程就会重复执行一次。

验证的过程，看这里的流程图：

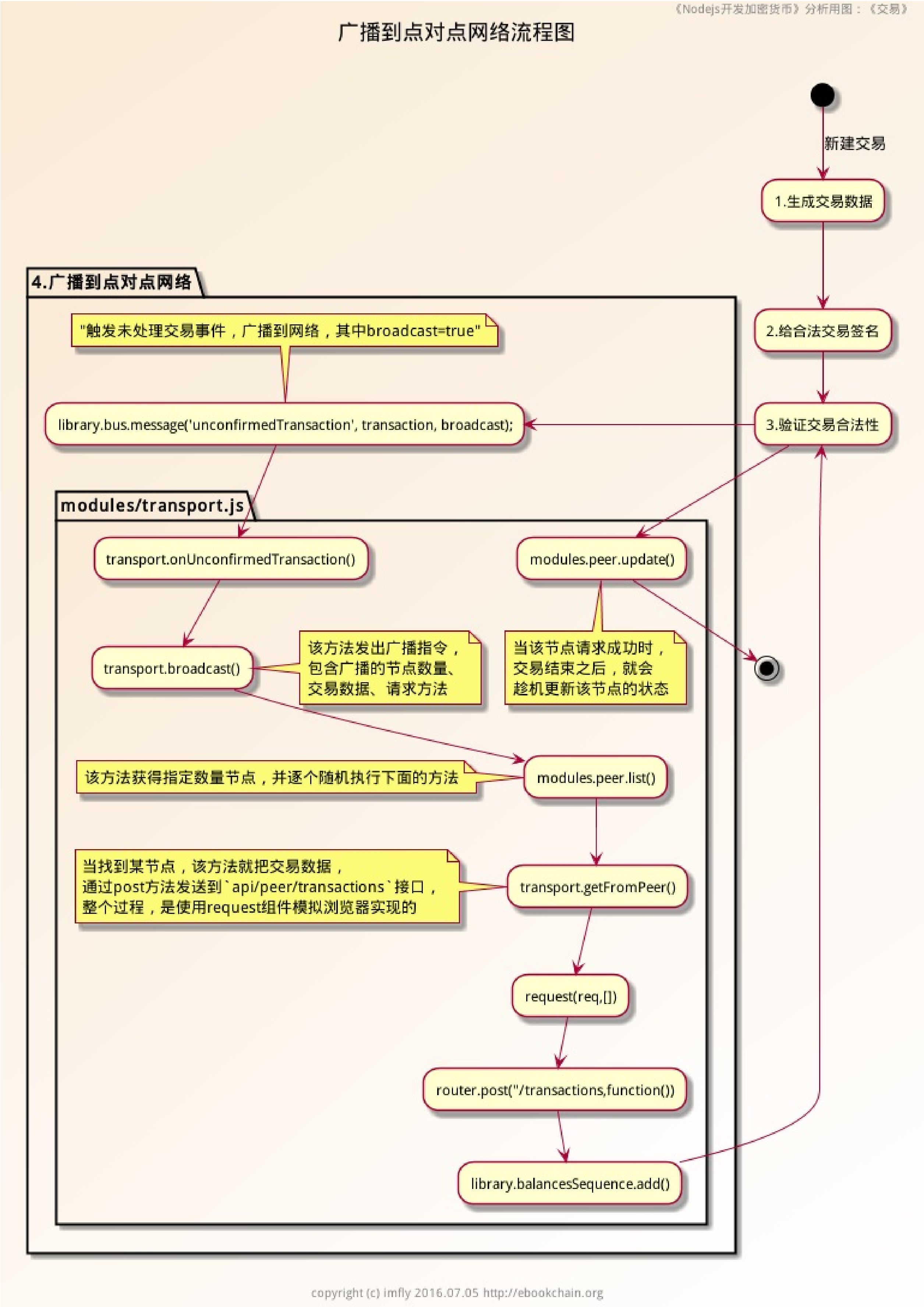

《Nodejs开发加密货币》分析用图：《交易》
验证交易合法性流程图
新建交易
1.生成交易数据
2.给合法交易签名
3.验证交易合法性
代码在modules.transactions.receiveTransactions()方法里，
显然，新建并签名的交易数据属于未确认交易
处理（未确认）交易：modules.transactions.receiveTransactions()
执行processUnconfirmedTransaction()方法
transaction.requesterPublicKey?
yes
no
library.logic.transaction.process(transaction, sender, requester,cb)
library.logic.transaction.process(transaction, sender,cb)
这个是转账交易类型，该函数本身也进行了系列验证，
并使用交易ID查询区块链数据库，确保交易属于未完成
这个是除转账之外的功能性交易类型，
如注册别名地址等
这里要避免“双花”问题
private.doubleSpendingTransactions[transaction.id]
library.logic.transaction.verify()
前面数据整理部分是对数据正确性的处理，
而该方法是对交易的验证，主要包含：验证
发送者（甲方）、签名、支付签名、签名唯
一性、交易费、数额、时间戳等，代码简单
加入区块链
private.addUnconfirmedTransaction()
把未确认的交易都放到一个队列
里处理，等同于下面的方法
transactions.prototype.applyUnconfirmed()
library.logic.transaction.applyUnconfirmed(transaction, sender, requester, cb);
通过该方法把交易信息关联到（加入）到区块链
this.scope.account.merge(sender.address,)
private.types[trs.type].applyUnconfirmed.call(this, trs, sender,)
4.广播到点对点网络
copyright (c) imfly 2016.07.05 http://ebookchain.org

（4）广播到点对点网络

没有中心服务器，必须借助点对点网络，把交易数据写入分布式公共账本——区块链，保证交易数据永远无法篡改，而且可以轻松查询追溯。这在中心化的服务器上，为了应对个别交易摩擦，保证交易记录可追溯，要采取更多的技术手段，记录更多的数据字段，意味着要保持大量数据冗余，付出更多资金成本。

因为交易数据不含私密信息，对网络没有苛刻要求，因此加密货币的网络可以覆盖很广，对网络的编程也变得灵活很多。理论上，只要能保证联通的便捷和快速，具体设计中不需要考虑更多复杂的因素。当然，就亿书这款产品而言，独有的用户协作和分享功能，对网络编程的性能有自身的要求，就另当别论，这方面将在下一个版本中体现出来。

这里，仅仅是加密货币基础网络功能，交易广播到网络的流程如下：


《Nodejs开发加密货币》分析用图：《交易》
广播到点对点网络流程图
新建交易
1.生成交易数据
2.给合法交易签名
3.验证交易合法性
4.广播到点对点网络
"触发未处理交易事件，广播到网络，其中broadcast=true"
library.bus.message('unconfirmedTransaction', transaction, broadcast);
modules/transport.js
transport.onUnconfirmedTransaction()
modules.peer.update()
transport.broadcast()
该方法发出广播指令，
包含广播的节点数量、
交易数据、请求方法
当该节点请求成功时，
交易结束之后，就会
趁机更新该节点的状态
该方法获得指定数量节点，并逐个随机执行下面的方法
modules.peer.list()
当找到某节点，该方法就把交易数据，
通过post方法发送到`api/peer/transactions`接口，
整个过程，是使用request组件模拟浏览器实现的
transport.getFromPeer()
request(req,[])
router.post("/transactions,function())
library.balancesSequence.add()
copyright (c) imfly 2016.07.05 http://ebookchain.org

转账交易分析

前面几篇，我们接触到几种交易类型，比如：注册别名地址和多重签名地址，不过并没有研究具体的交易过程，下面通过分析 `转账交易` 来学习整个交易、验证的过程。

代码实现在 `modules/transactions.js` 文件里，主要Api如下：

```
// 148行
router.map(shared, {
  "get /": "getTransactions",
  "get /get": "getTransaction",
  "get /unconfirmed/get": "getUnconfirmedTransaction",
  "get /unconfirmed": "getUnconfirmedTransactions",
  "put /": "addTransactions"
});

// 160行
library.network.app.use('/api/transactions', router);
```

解析一下，就是：

```
get /api/transactions/ -> shared.getTransactions
get /api/transactions/get -> shared.getTransaction
get /api/transactions/unconfirmed/get -> shared.getUnconfirmedTransaction
get /api/transactions/unconfirmed -> shared.getUnconfirmedTransactions
put /api/transactions/ -> shared.addTransactions
```

我们仍然把读取数据的Api放一放，因为他们很简单，重点掌握写数据的操作， `put /api/transactions/` ，对应方法 `shared.addTransactions` ，代码如下：

```
// 652行
shared.addTransactions = function (req, cb) {
    var body = req.body;
    library.scheme.validate(body, {
        type: "object",
        properties: {
            secret: {
                type: "string",
                minLength: 1,
                maxLength: 100
            },
            amount: {
                type: "integer",
                minimum: 1,
                maximum: constants.totalAmount
            },
            recipientId: {
                type: "string",
                minLength: 1
            },
            publicKey: {
```

```
                type: "string",
                format: "publicKey"
            },
            secondSecret: {
                type: "string",
                minLength: 1,
                maxLength: 100
            },
            multisigAccountPublicKey: {
                type: "string",
                format: "publicKey"
            }
        },
        //
        required: ["secret", "amount", "recipientId"]
    }, function (err) {
        // 验证数据格式
        if (err) {
            return cb(err[0].message);
        }

        // 验证密码信息
        var hash = crypto.createHash('sha256').update(body.secret, 'utf8').digest();
        var keypair = ed.MakeKeypair(hash);

        if (body.publicKey) {
            if (keypair.publicKey.toString('hex') != body.publicKey) {
                return cb("Invalid passphrase");
            }
        }

        var query = {};

        // 乙方（接收方）地址转换，保证可以用户名转账
        var isAddress = /^[0-9]+[L|l]$/g;
        if (isAddress.test(body.recipientId)) {
            query.address = body.recipientId;
        } else {
            query.username = body.recipientId;
        }

        library.balancesSequence.add(function (cb) {
            // 验证乙方用户合法性
            modules.accounts.getAccount(query, function (err, recipient) {
                if (err) {
                    return cb(err.toString());
                }
                if (!recipient && query.username) {
                    return cb("Recipient not found");
                }

                var recipientId = recipient ? recipient.address : body.recipientId;
                var recipientUsername = recipient ? recipient.username : null;
```

```
                // 验证甲方（发送方）用户合法性
                if (body.multisigAccountPublicKey && body.multisigAccountPublicKey !=
keypair.publicKey.toString('hex')) {
                    // 验证多重签名
                    modules.accounts.getAccount({publicKey: body.multisigAccountPublic
Key}, function (err, account) {
                        if (err) {
                            return cb(err.toString());
                        }
                        // 多重签名帐号不存在
                        if (!account || !account.publicKey) {
                            return cb("Multisignature account not found");
                        }
                        // 多重签名帐号未激活
                        if (!account || !account.multisignatures) {
                            return cb("Account does not have multisignatures enabled")
;
                        }
                        // 帐号不属于该多重签名组
                        if (account.multisignatures.indexOf(keypair.publicKey.toString
('hex')) < 0) {
                            return cb("Account does not belong to multisignature group
");
                        }

                        // 接着验证甲方（发送方）用户合法性
                        modules.accounts.getAccount({publicKey: keypair.publicKey}, fu
nction (err, requester) {
                            if (err) {
                                return cb(err.toString());
                            }
                            // 甲方帐号不存在
                            if (!requester || !requester.publicKey) {
                                return cb("Invalid requester");
                            }

                            // 甲方支付密码（二次签名）不正确
                            if (requester.secondSignature && !body.secondSecret) {
                                return cb("Invalid second passphrase");
                            }

                            // 甲方帐号公钥与多重签名帐号公钥是不一样的（因为两个账户是不一样的）
                            if (requester.publicKey == account.publicKey) {
                                return cb("Invalid requester");
                            }

                            var secondKeypair = null;

                            if (requester.secondSignature) {
                                var secondHash = crypto.createHash('sha256').update(bo
dy.secondSecret, 'utf8').digest();
                                secondKeypair = ed.MakeKeypair(secondHash);
```

```
                              }

                              try {
                                  // 763行 把上述数据整理成需要的交易数据结构，并给交易添加时间戳
、签名、生成ID、计算交易费等
                                  var transaction = library.logic.transaction.create({
                                      type: TransactionTypes.SEND,
                                      amount: body.amount,
                                      sender: account,
                                      recipientId: recipientId,
                                      recipientUsername: recipientUsername,
                                      keypair: keypair,
                                      requester: keypair,
                                      secondKeypair: secondKeypair
                                  });
                              } catch (e) {
                                  return cb(e.toString());
                              }

                              // 776行 处理交易
                              modules.transactions.receiveTransactions([transaction], cb
);
                          });
                      });
                  } else {
                      // 直接验证甲方（发送方）用户合法性，这里的请求者requester就是发出交易者send
er
                      ...
    });
}
```

上面这段代码涉及到的就是生成交易数据，这与之前的《地址》、《签名和多重签名》里提到的功能性交易差不多，这里把该方法代码完整粘贴出来，具体逻辑请看代码里的注释和前面的流程图。

接下来，776行，通过 `receiveTransactions` 方法处理交易，该方法最终调用的是下面的方法。关键部分，已经添加了注释，请结合上面的流程图阅读，不再详述。

```
// modules/transactions.js文件
// 337行
Transactions.prototype.processUnconfirmedTransaction = function (transaction, broadcas
t, cb) {
    modules.accounts.setAccountAndGet({publicKey: transaction.senderPublicKey}, functi
on (err, sender) {
        // 这是个闭包，在下面的程序运行结束的时候才调用，因此是验证完毕，才写入区块链、广播到网络
        function done(err) {
            if (err) {
                return cb(err);
            }
            // 这里 加入区块链 操作
            privated.addUnconfirmedTransaction(transaction, sender, function (err) {
```

```
                if (err) {
                    return cb(err);
                }
                // 触发事件，广播到网络
                library.bus.message('unconfirmedTransaction', transaction, broadcast);

                cb();
            });
        }

        if (err) {
            return done(err);
        }

        if (transaction.requesterPublicKey && sender && sender.multisignatures && send
er.multisignatures.length) {
            modules.accounts.getAccount({publicKey: transaction.requesterPublicKey}, f
unction (err, requester) {
                if (err) {
                    return done(err);
                }

                if (!requester) {
                    return cb("Invalid requester");
                }
                // 开始执行一系列验证，包括交易是不是已经存在
                library.logic.transaction.process(transaction, sender, requester, func
tion (err, transaction) {
                    if (err) {
                        return done(err);
                    }

                    // 检查是否交易已经存在（包括双花交易）
                    if (privated.unconfirmedTransactionsIdIndex[transaction.id] !== un
defined || privated.doubleSpendingTransactions[transaction.id]) {
                        return cb("Transaction already exists");
                    }
                    // 这里是 直接验证交易签名等信息，接着调用闭包 done()，把交易写入区块链并广播
到网络
                    library.logic.transaction.verify(transaction, sender, done);
                });
            });
        } else {
            ...
}
```

总结

这里的编码逻辑非常清晰，但作为非常核心的部分，使用了大量编程技巧，需要比较熟练的开发技能。代码中涉及到大量的回调和验证，有的回调嵌套很深，需要对异步较为深入的理解，掌握熟练的回调处理方法，不然理解和编码都会有很多困扰。因此，好好熟悉基本编码技巧，从小处着手打好基础很重要。

本文涉及的流程图相对比较复杂，为了印刷方便，我把完整的流程图折分成为四张，处理过程花费了大量时间，但是很多细节仍然无法照顾到，也无法保证没有错误和疏漏，请看到问题的小伙伴及时反馈给我。

交易是怎么写入区块链的，上面仅仅点到为止，不够详细和深入。为了进一步阐述区块链的原理，需要专门拿出一篇来，详细讲述。而且，作为目前加密货币的“网红”，区块链也值得我们好好研究。请看下一篇：《神秘的区块链》

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org

亿书白皮书： http://ebookchain.org/ebookchain.pdf

亿书官网： http://ebookchain.org

亿书官方QQ群：185046161（亿书完全开源开放，欢迎各界小伙伴参与）

参考

精通比特币（英文）

精通比特币（中文）

区块链

前言

亿书，是一款加密货币产品，用时髦的话说，更是一款实用的区块链产品。那么，区块链是什么么？有那些特点？最近，以太坊硬分叉事件给了我们很多启示，能不能彻底杜绝区块链分叉行为？这一章，我们通过认真阅读和理解亿书相关的代码逻辑，来详细解释和说明这些问题，以便更加深入的了解和学习这项技术。

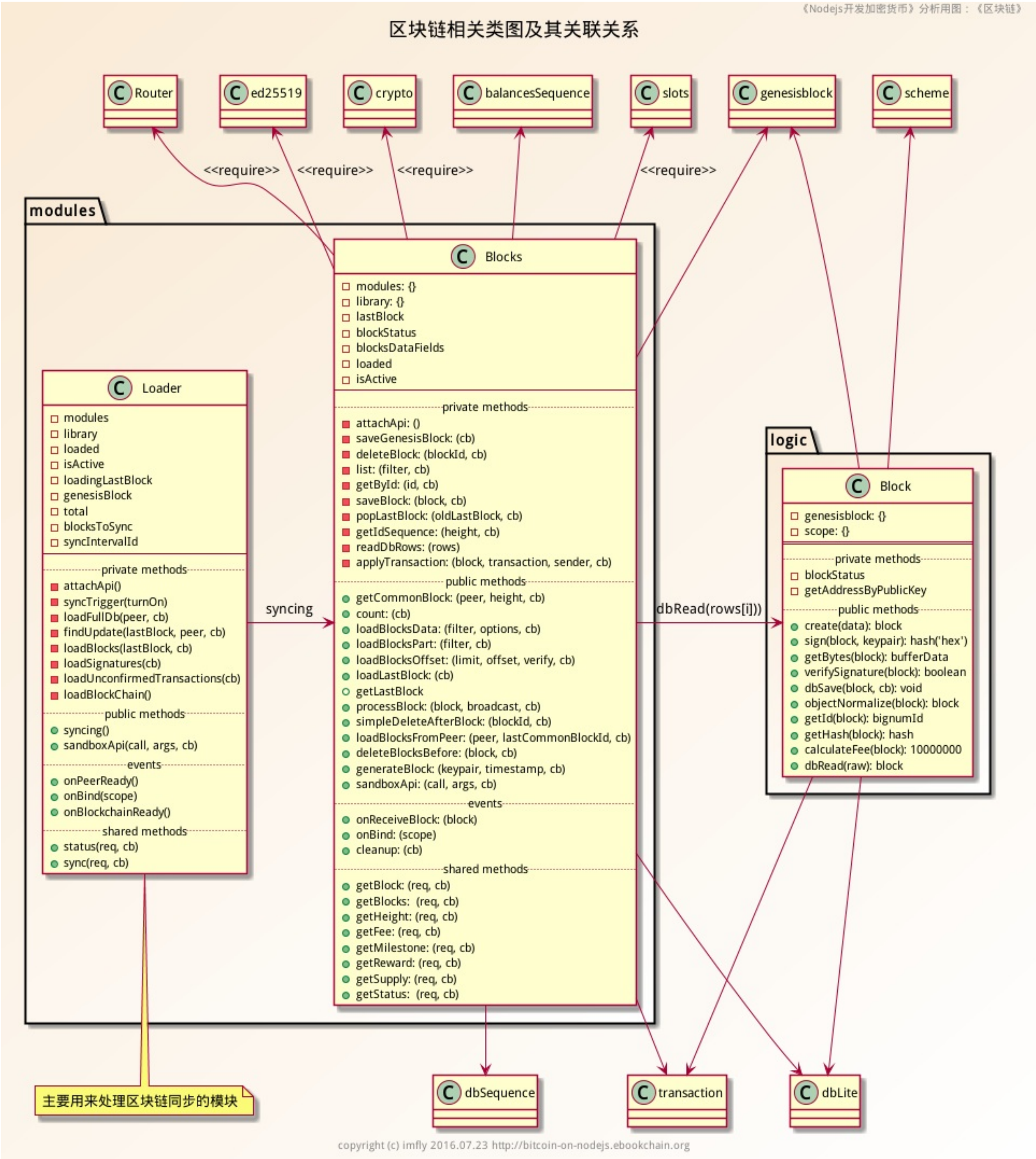
源码

blocks.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/blocks.js

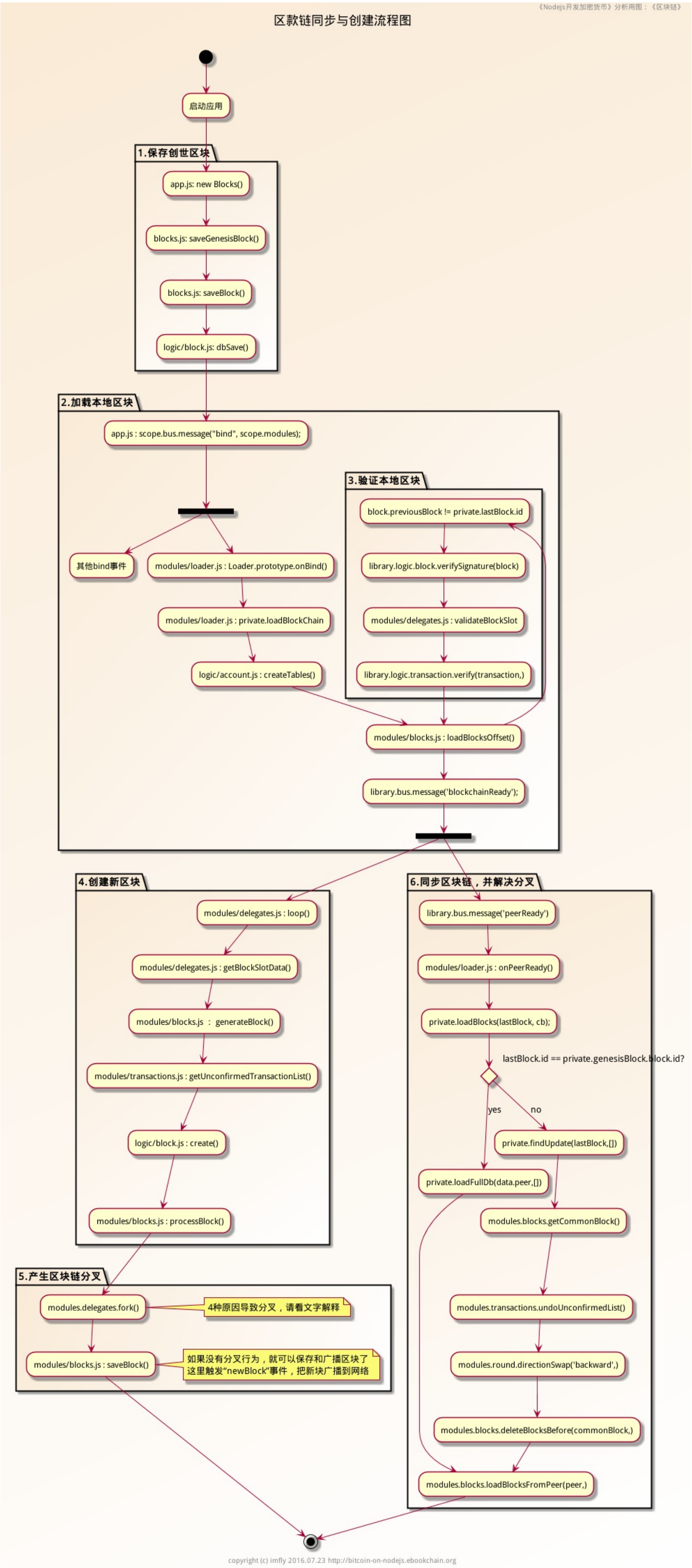
block.js https://github.com/Ebookcoin/ebookcoin/blob/logic/block.js

loader.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/loader.js

类图


《Nodejs开发加密货币》分析用图：《区块链》
区块链相关类图及其关联关系
Router
ed25519
crypto
balancesSequence
slots
genesisblock
scheme
<<require>>
<<require>>
<<require>>
<<require>>
modules
Blocks
modules: {}
library: {}
lastBlock
blockStatus
blocksDataFields
loaded
isActive
private methods
attachApi: ()
saveGenesisBlock: (cb)
deleteBlock: (blockId, cb)
list: (filter, cb)
getById: (id, cb)
saveBlock: (block, cb)
popLastBlock: (oldLastBlock, cb)
getIdSequence: (height, cb)
readDbRows: (rows)
applyTransaction: (block, transaction, sender, cb)
public methods
getCommonBlock: (peer, height, cb)
count: (cb)
loadBlocksData: (filter, options, cb)
loadBlocksPart: (filter, cb)
loadBlocksOffset: (limit, offset, verify, cb)
loadLastBlock: (cb)
getLastBlock
processBlock: (block, broadcast, cb)
simpleDeleteAfterBlock: (blockId, cb)
loadBlocksFromPeer: (peer, lastCommonBlockId, cb)
deleteBlocksBefore: (block, cb)
generateBlock: (keypair, timestamp, cb)
sandboxApi: (call, args, cb)
events
onReceiveBlock: (block)
onBind: (scope)
cleanup: (cb)
shared methods
getBlock: (req, cb)
getBlocks: (req, cb)
getHeight: (req, cb)
getFee: (req, cb)
getMilestone: (req, cb)
getReward: (req, cb)
getSupply: (req, cb)
getStatus: (req, cb)
Loader
modules
library
loaded
isActive
loadingLastBlock
genesisBlock
total
blocksToSync
syncIntervalId
private methods
attachApi()
syncTrigger(turnOn)
loadFullDb(peer, cb)
findUpdate(lastBlock, peer, cb)
loadBlocks(lastBlock, cb)
loadSignatures(cb)
loadUnconfirmedTransactions(cb)
loadBlockChain()
public methods
syncing()
sandboxApi(call, args, cb)
events
onPeerReady()
onBind(scope)
onBlockchainReady()
shared methods
status(req, cb)
sync(req, cb)
syncing
dbRead(rows[i]))
logic
Block
genesisblock: {}
scope: {}
private methods
blockStatus
getAddressByPublicKey
public methods
create(data): block
sign(block, keypair): hash('hex')
getBytes(block): bufferData
verifySignature(block): boolean
dbSave(block, cb): void
objectNormalize(block): block
getId(block): bignumId
getHash(block): hash
calculateFee(block): 10000000
dbRead(raw): block
主要用来处理区块链同步的模块
dbSequence
transaction
dbLite
copyright (c) imfly 2016.07.23 http://bitcoin-on-nodejs.ebookchain.org

流程图


《Nodejs开发加密货币》分析用图：《区块链》
区款链同步与创建流程图
启动应用
1.保存创世区块
app.js: new Blocks()
blocks.js: saveGenesisBlock()
blocks.js: saveBlock()
logic/block.js: dbSave()
2.加载本地区块
app.js : scope.bus.message("bind", scope.modules);
3.验证本地区块
block.previousBlock != private.lastBlock.id
library.logic.block.verifySignature(block)
modules/delegates.js : validateBlockSlot
library.logic.transaction.verify(transaction,)
其他bind事件
modules/loader.js : Loader.prototype.onBind()
modules/loader.js : private.loadBlockChain
logic/account.js : createTables()
modules/blocks.js : loadBlocksOffset()
library.bus.message('blockchainReady');
4.创建新区块
modules/delegates.js : loop()
modules/delegates.js : getBlockSlotData()
modules/blocks.js : generateBlock()
modules/transactions.js : getUnconfirmedTransactionList()
logic/block.js : create()
modules/blocks.js : processBlock()
6.同步区块链，并解决分叉
library.bus.message('peerReady')
modules/loader.js : onPeerReady()
private.loadBlocks(lastBlock, cb);
lastBlock.id == private.genesisBlock.block.id?
yes
no
private.findUpdate(lastBlock,[])
private.loadFullDb(data.peer,[])
modules.blocks.getCommonBlock()
modules.transactions.undoUnconfirmedList()
modules.round.directionSwap('backward',)
modules.blocks.deleteBlocksBefore(commonBlock,)
modules.blocks.loadBlocksFromPeer(peer,)
5.产生区块链分叉
modules.delegates.fork()
4种原因导致分叉，请看文字解释
modules/blocks.js : saveBlock()
如果没有分叉行为，就可以保存和广播区块了
这里触发"newBlock"事件，把新块广播到网络
copyright (c) imfly 2016.07.23 http://bitcoin-on-nodejs.ebookchain.org

解读

1.区块链是什么？

这里的区块链指狭义的区块链，是一种自引用的数据结构，可以存储成文件形式，不过多数产品存储在一个数据库中，比如比特币使用Google的LevelDB数据库存储。详细的概念解析，请看第一部分《区块链架构设计简介》，有助于更好的理解区块链概念。

（1）从数据库设计角度理解区块链

用数据库的概念理解，区块链就是一张“自引用”的数据库表。每条记录代表一个区块，这条记录（区块）记录着它前面（时间上）一条记录的信息，可以直接查询到前一条记录，因此从任何一条记录开始都可以往前顺序追溯，直到第一条记录。普通自引用表结构，通常使用ID作为关联外键，加密货币使用的是经过加密处理的信息字段，具有签名认证作用，可实现自我验证，防止被篡改。

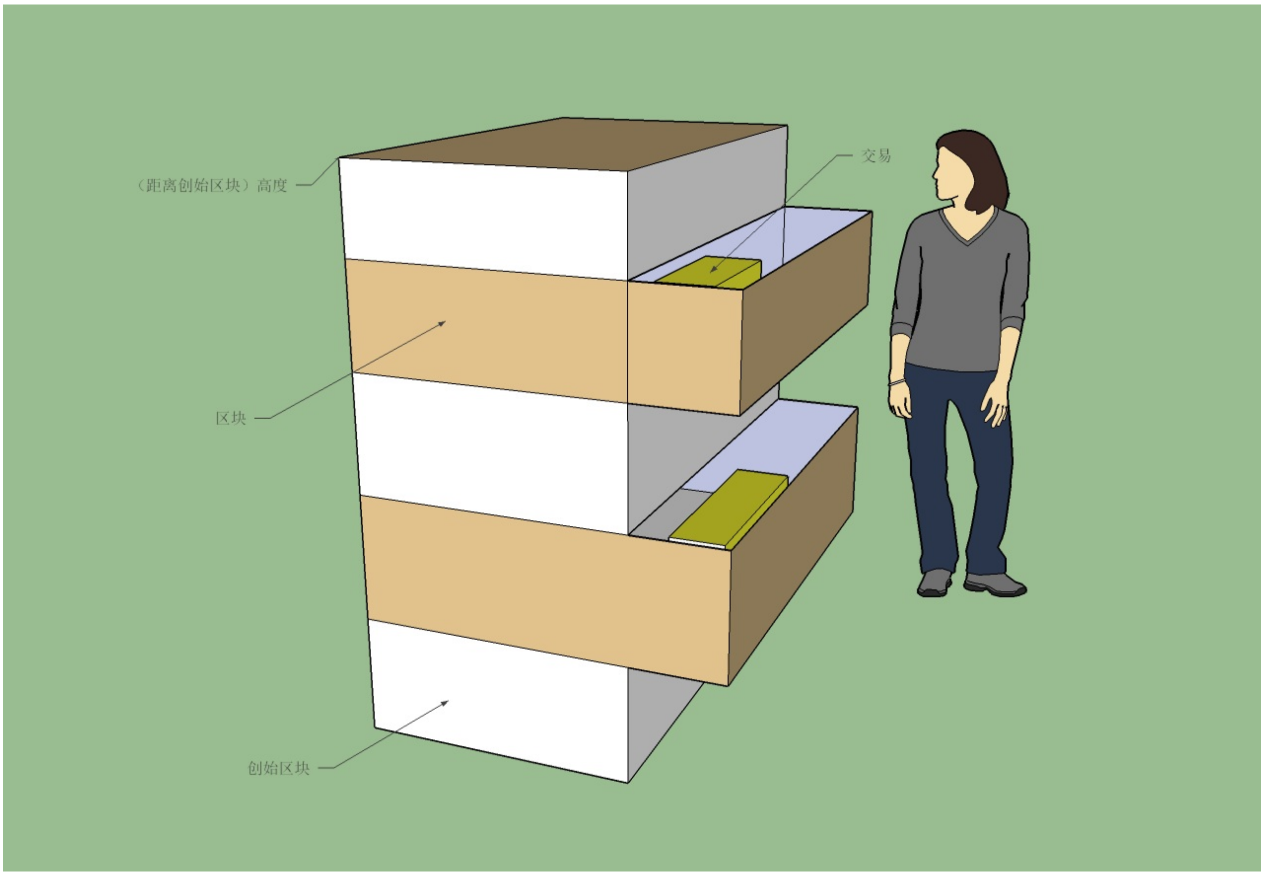
与区块链直接关联的另一张重要的表，就是交易表。加密货币包含大量的交易，我们之前分析过，交易可以是加密货币，也可以是债权、股权或版权等各类数字资产，这些交易保存在一张独立表里，并与区块链形成多对一的关联方式。如此以来，只要追溯到区块，就很容易查询到该区块包含的交易记录。这样，一个公开透明、无法篡改、方便追溯的账本就形成了。

上面是从数据库查询（读数据）的角度考虑的，如果从写入的角度思考，就更有意思了。写入是要根据需求不同进行不同的编码，我们前面的章节说过，加密货币的各种功能都可以通过扩展交易类型进行编码，如果把一些现实中的合同规则进行编码，要求系统在某个条件下自动执行（写入或更新）某交易，自然也是件简单轻松的事情，这就是“智能合约”的简单理解。

我们在第一部分提到过“智能合约”的概念，在这里再次提及，作为程序员能够更加直观的去理解编码上的可行性。“智能合约”也是目前加密货币社区讨论火热的概念，可以发挥你想象的翅膀，从加密货币扩展到现实世界的各种场景，比如：自动贩卖机、销售终端、大公司间的电子数据交换、银行间用于转移与清算的支付网络，以及音乐、电影和电子书等数字版权交易等。

（2）形象化理解区块链

人们通常把具有先后顺序的数据结构，使用栈来表示，比特币白皮书把这种结构进一步形象化，第一个区块作为栈底，然后其他区块按照时间顺序依次堆叠在上面，这样一来，区块与首区块之间的距离就表示“高度”，“顶端”就表示最新添加的区块。每个区块包含大量交易，就是包含在对应栈里的数据。我们可以把这样的结构想象成一个大大的橱柜，区块就是其中一个抽屉，每个抽屉里是满满的交易，就象下面这样：



（3）区块链分叉

物理分叉。每一个区块都与它的前一区块（父区块）关联，而对它后面的区块（子区块）无限制，也就是说最顶端的区块，肯定知道它的父区块（已经写入区块链），但不知道子区块（或许还没有产生，也可能在传输的过程中）。我们知道，从物理层面，数据库、硬盘、网络的IO操作是最耗费时间的，在某一个时刻，多个最新区块同时找到父区块是很常见的现象，这就必然导致区块链分叉（从主链向多个方向发展）。这是在同一个软件版本（及其兼容版本）的情况下发生的，没有人为干预，不妨叫做物理分叉。

显然，物理分叉取决于物理环境，这与什么样的共识机制没有直接关系，不论是采取工作量证明机制（PoW)的比特币、还是采取股权证明机制（PoS）的点点币，亦或是这里采取授权股权证明机制（DPoS）的亿书币，都是如此。为了保持区块链的单一链条，解决分叉的最简单方式就是放任每个分叉继续增长，通常在下一刻就会出现差别，这时候软件选择最长的那个链条作为主链即可。在具体的设计开发过程中，这也是一个逻辑相对复杂的难点。

人为分叉。那么，如果存在人的干预，会怎么样呢？我们知道，世界上没有绝对完美的东西，人类开发设计的软件也不例外，漏洞是常有的，看看微软的windows系统时不时跳出来的漏洞修复提醒，就知道这类事情多么常见。而且，人类的需求始终在变化，软件要不断推出新功能来应对。所以，软件出现漏洞，或者添加新的功能，这类情况是再正常不过的事情。这时候，旧版本的软件对新版本软件产生的区块可能出现兼容性问题，甚至需要人为改变区块链的走向，这就导致新旧版本之间出现分叉，不妨叫做人为分叉。

很显然，人为分叉也是无法避免的事情。你可能认为很简单，有漏洞就修复吧，有新功能加上就得了，有什么好解释的。事实上，加密货币核心是交易，是价值转移的手段，规则的改变直接关系到所有持币人的利益。我们在第一部分说过，人是趋利的，要追求利益最大化，新功能能否保护用户的利益，还是代表了少部分利益集团的意志，应该如何约束和决策，这已经不单单是一个技术问题，更多的是社区政治问题，需要社区共同参与。历史证明，承载了较大资金盘的加密货币，在某一次分叉过程中，个别用户或矿工没有及时更新软件，就造成了直接经济损失。所以，每一个持币用户都非常关心任何一次分叉行为，都有可能站出来表达自己的意愿，甚至选择留在旧链上。

硬分叉和软分叉。它们都属于人为分叉，孰优孰劣也是当前社区分歧比较严重的问题。最初，社区区分这两个概念的简单方法就是，“硬分叉”是与旧版本的兼容度不高，但是获得了社区共识的规则明确的分叉行为，“软分叉”恰恰相反。发展到今天，只要是明确的“分叉”行为，大家都会寻求社区共识，所以二者的区别主要集中在软件兼容问题上了。这里的社区共识，是指包括软件开发者、矿工和使用者在内的整个软件社区，采取投票等方式，获得最大程度的一致性意见，通常是90%以上的社区成员同意就认为是达成了社区共识。比如最近以太坊为了应对The Dao遭受黑客攻击而实施的紧急硬分叉，就获得了社区87%的同意（接近90%，这里不讨论这次分叉行为的好坏）。

从技术角度讲，这里所谓的“硬”，主要体现在与旧版本的不兼容（或少量兼容）上，属于抛弃旧版本的行为，如果用户不升级软件，就永远留在旧链上，感觉上更加强硬得多。“软分叉”，最大程度的保持了对以前版本的兼容性，做得好的话，多个版本可以同时运行，类似于正常的版本迭代升级，用户可以自由选择是否升级。有人说“硬分叉”是很糟糕的事情，另一些人认为“软分叉”风险更大。事实证明，只要准备充分，操作得当，无论是“硬分叉”，还是“软分叉”，都不可怕，只不过“软分叉”在编码中需要考虑的情况更加复杂，对用户的影响较为隐蔽。

我个人不喜欢谈政治，但是作为交易媒介的新兴产物——加密货币，天生就是政治的附属品。这些货币的持续发展，往往是不同利益团体（包括开发者、矿工和用户）之间不断博弈的结果。最初是开发者主导，某个时期矿工的力量更加强大，后期用户的力量就不容忽视。也正是这些力量之间的制衡，才让加密货币相对持续稳健的发展，当这三种力量达到均衡的时候，就是这个加密货币相对成熟的时候。最近，以太坊硬分叉处理Dao遭受的黑客攻击事件，搅动了整个加密货币社区，不同利益者发出不同的声音，这是好事，充分说明加密货币仍处在初级阶段，还有很长的路要走。

2.区块链的特点

我们可以按照堆栈的方式理解数据结构，并采用自引用的关联方式设计数据库模型，但是做到这些，我个人认为并不代表就是区块链了，它还必须使用加解密技术，被置于去中心化的网络，由P2P网络节点共同维护，才能称得上区块链。当然，也有人持不同观点，他们认为一个中心化的应用，如果使用类似的数据结构，会更加安全（比不使用该结构的中心化系统），同时可以避免分叉，性能或许更高（比去中心化的系统），但事实上没有了P2P网络的支撑，这点改进算不上什么。我们之前分析过，P2P网络本身就是一条非常好的安全屏障，单

点被攻击或被破解，对整个网络系统没有太大伤害，而任何中心化的系统仅仅相当于单节点，安全性大大降低。所以，为了那些许的性能改进，却要牺牲更好的安全性，有点得不偿失。

汇总以上信息，区块链应该具备这样几个特点：

- 分布存储：区块链处于P2P网络之中，无论什么公链、私链，还是联盟链，都要采取分布式存储，使用一种机制保证区块链的同步和统一;
- 公开透明：每个节点都有一个区块链副本，区块链本身没有加密，数据可以任意检索和查询，甚至可以修改（改了也没用）;
- 无法篡改：这是加密技术的巧妙应用，每一区块都会记录前一区块的信息，并实现验证，确保无法篡改。这里的无法篡改不是不能改，而是局部修改的数据，无法通过验证，要想通过验证，必须修改整个区块链，这在理论上可行，操作上不可行;
- 方便追溯：区块链是公开的，从任一区块都可以向前追溯，直到第一个区块，并通过区块查到与之关联的全部交易；
- 存在分叉：这是由P2P网络等物理环境，以及软件开发实践过程决定的，人们无法根本性杜绝。

也正是因为这样的特点，区块链的概念才逐渐火爆起来。实践证明，区块链技术能实现一切中心化应用的场景，可以解决（或更好的解决）很多中心化应用无法解决的问题，比如分布式财务管理、分布式存储、知识产权保护、电子商务，乃至物联网，特别是对于金融业而言，资金清算、审计等等，成本会大幅度降低。亿书，就是利用它公开透明、可追溯的特点，与数字出版结合起来，实现自媒体和版权保护，彻底解决当前数字出版版权保护不力的顽疾。

3.区块链开发应该解决的问题

明白区块链是什么和基本原理之后，就可以着手设计其基本功能了。从需求的角度说，设计中需要做到如下几点：

（1）加载区块链。确保本地区块链合法，未被篡改。

- 保存创世区块
- 加载本地区块
- 验证本地区块

（2）处理新区块。加载后，该节点就可以处理网络中的交易了。

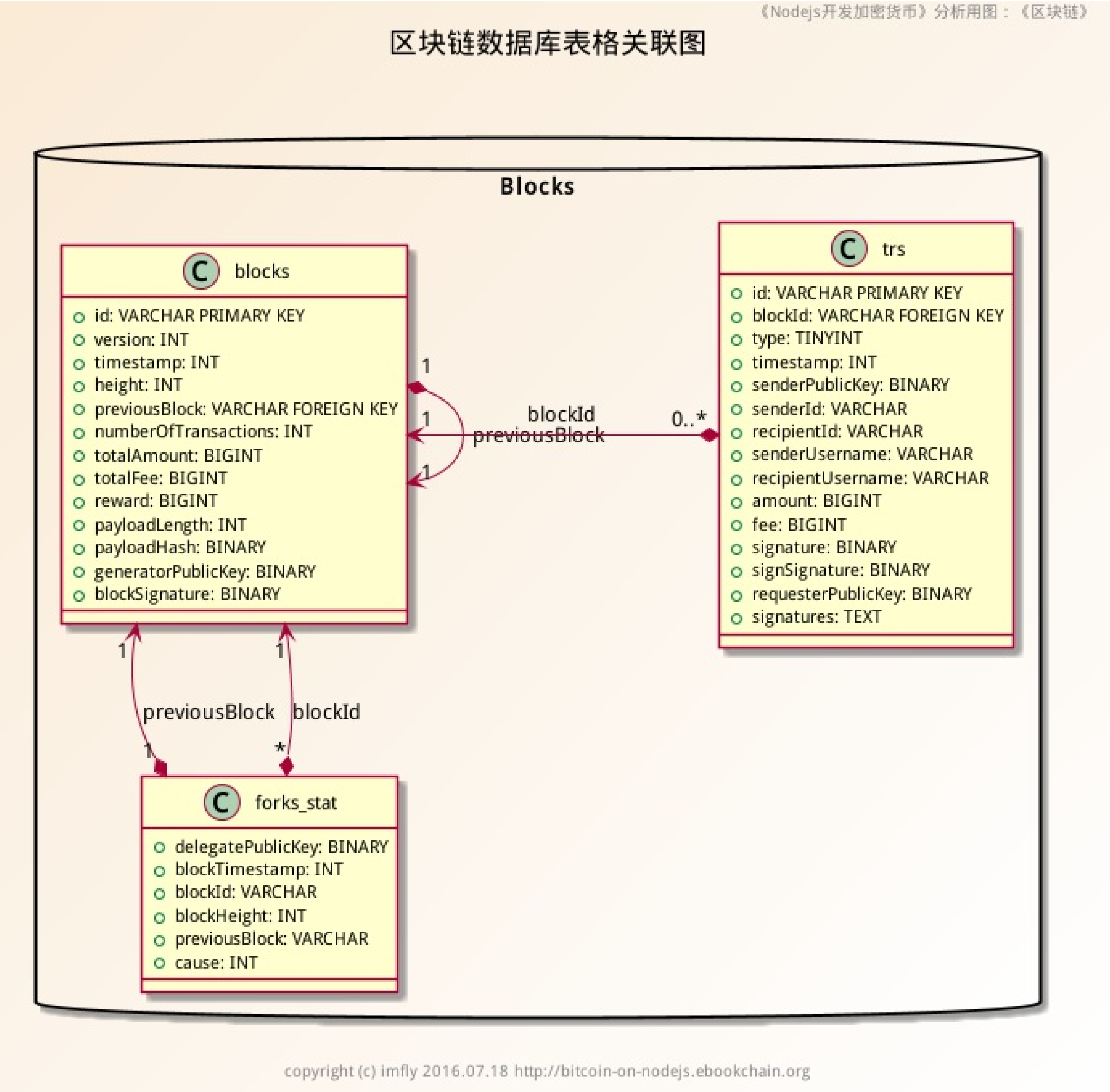
- 创建新区块；
- 收集整理交易，写入（关联）区块；
- 把新产生的区块写入区块链；
- 处理区块链分叉。

（3）同步区块链。确保本地区块链与网络中完整的区块链同步。

下面，我们从数据库设计出发，分别研读相关代码，认真讨论亿书区款链是如何运作的。

4.亿书区块链数据库设计

亿书使用SQLite数据库，与区块链相关的数据库结构如图：



blocks表是区块链，trs表是各种交易，forks_stat表代表分叉状态。从关联关系上看，blocks首先是一个自引用表，使用previousBlock关联；与trs是一对多的关系，一条记录关联多条交易；与forks_stat也是一对多的关系，意思是有分叉。

5.亿书区块链实现

这里按照上面提到的开发区块链要解决的问题，逐一对照，查看亿书技术实现。

（1）保存创世区块

创世区块是硬编码到客户端程序里的，会在客户端运行的时候，直接写入数据库。这样做的好处是保证每个客户端都有一个安全、可信的区块链的根。

```
// modules/blocks.js
// 78行
function Blocks(cb, scope) {
    library = scope;
    // 80行
    genesisblock = library.genesisblock;
    self = this;
    self.__private = privated;
    privated.attachApi();

    // 85行
    privated.saveGenesisBlock(function (err) {
        setImmediate(cb, err, self);
    });
}
```

这是`modules/blocks.js`模块的构造函数，在入口程序`app.js`运行的时候，直接创建该模块的实例，85行的代码`privated.saveGenesisBlock`方法直接运行。如果已经运行过，该方法就会返回，什么都不做。如果第一次运行，该方法就会直接保存创世区块（80行的genesisblock），接着调用266行的`privated.saveBlock()`方法（不再粘贴），把创世区块记录（包括交易）保存到数据库。

这是一个非常典型的区块创建过程，我们可以借机会看看一个区块（创世）的数据是什么样的：

```
// genesisBlock.json 文件
{
  "version": 0,

    // 3行
  "totalAmount": 10000000000000000,
  "totalFee": 0,
  "reward": 0,
  "payloadHash": "1cedb278bd64b910c2d4b91339bc3747960b9e0acf4a7cda8ec217c558f429ad",
  "timestamp": 0,
  "numberOfTransactions": 103,
  "payloadLength": 20326,
  "previousBlock": null,
  "generatorPublicKey": "b7b46c08c24d0f91df5387f84b068ec67b8bfff8f7f4762631894fce4aff6
c75",

    // 1757行
  "height": 1,
  "blockSignature": "2985d896becdb91c283cc2366c4a387a257b7d4751f995a81eae3aa705bc24fdb
950c3afbed833e7d37a0a18074da461d68d74a3a223bc5f8e9c1fed2f3fec0e",
  "id": "8593810399212843182"，
```

```
    // 12行。为了方便阅读，这里把关联的交易信息排版在最后位置
    "transactions": [
    {
      "type": 0,

            // 15行
      "amount": 10000000000000000,
      "fee": 0,
      "timestamp": 0,
      "recipientId": "6722322622037743544L",
      "senderId": "5231662701023218905L",
      "senderPublicKey": "b7b46c08c24d0f91df5387f84b068ec67b8bfff8f7f4762631894fce4aff
6c75",
      "signature": "aa413208c32d00b89895049ff21797048fa41c1b2ffc866900ffd97570f8d87e85
2c87074ed77c6b914f47449ba3f9d6dca99874d9f235ee4c1c83d1d81b6e07",
      "id": "5534571359943011068"
    },
        {
            "type": 2,
            ...
        },

        ...

        {
      "type": 3,
      ...
    }
    ...
  ]
}
```

这些字段，我们在上面的数据库表里已经列出，下面看看几个关键数据：

3行：在创世区块设定初始代币总量，这里是1亿; 1757行：创世区块高度为1; 12行：区块必须包含交易，这里是3种类型的交易，之前分析过，它们分别是转账交易、受托人交易和投票交易。第1个转账交易，把初始区块的代币全部转到了另一个账户，这在实际的生产环境，特别是在ICO（预售）之后，可以直接转给参与众筹的实际用户。所以，创世区块有其非常实际的意义。其它两种交易，是支撑亿书共识机制的。

（2）加载本地区块

任何节点，都需要先加载验证本地区块链，确保没有被篡改。这个加载过程是软件初始化过程中的一部分，开发中不需要与网络节点联网等其他问题纠缠在一起。因此，代码需要被放在入口文件中去执行。我们在《入口程序app.js解读》一章里解读了app.js文件，但并不详细，仅仅梳理了程序的大致流程。这里重新提及，专注于区块链加载验证的问题，这在具体开发过程中，是很正常的增量开发的思路。

```
// app.js文件
...
ready: ['modules', 'bus', function (cb, scope) {
    // 435行
    scope.bus.message("bind", scope.modules);
    cb();
}]
```

app.js文件 435行: 触发了“bind”事件（这是自定义的事件处理机制，请参考开发实践中关于事件循环的部分章节），会执行所有模块里的“onBind()”方法。该方法运行之前，各个模块仅仅被实例化，处于待命状态，所以“bind”事件是激活各模块的重要事件，是继各模块构造函数运行之后的关键方法（具体流程，请参考本部分第一章，模块的加载流程图）。大部分模块里的“onBind()”方法仅仅用来初始化某个变量，唯独 `loader.js` 模块，执行了下面的代码：

```
// modules/loader.js文件
Loader.prototype.onBind = function (scope) {
    modules = scope;
    // 534行
    privated.loadBlockChain();
};
```

modules/loader.js文件 534行： `privated.loadBlockChain()` 方法就是用来加载区块链的，内容如下：

```
// modules/loader.js文件
privated.loadBlockChain = function () {
    var offset = 0, limit = library.config.loading.loadPerIteration;
    var verify = library.config.loading.verifyOnLoading;

    // 357行，闭包
    function load(count) {
        verify = true;
        privated.total = count;

        library.logic.account.removeTables(function (err) {
            if (err) {
                throw err;
            } else {
                library.logic.account.createTables(function (err) {
                    if (err) {
                        throw err;
                    } else {
                        // 369行
                        async.until(
                            function () {
                                return count < offset;
                            }, function (cb) {
                                library.logger.info('Current ' + offset);
```

```
                                  setImmediate(function () {
                                      modules.blocks.loadBlocksOffset(limit, offset, ver
ify, function (err, lastBlockOffset) {
                                          if (err) {
                                              return cb(err);
                                          }
                                          // 380行
                                          offset = offset + limit;
                                          privated.loadingLastBlock = lastBlockOffset;

                                          cb();
                                      });
                                  });
                              }, function (err) {
                                    ...
                                      // 398行
                                      library.logger.info('Blockchain ready');
                                      library.bus.message('blockchainReady');
                                  }
                              }
                      ...
    }

    // 408行
    library.logic.account.createTables(function (err) {
        if (err) {
            throw err;
        } else {
            library.dbLite.query("select count(*) from mem_accounts where blockId = (s
elect id from blocks where numberOfTransactions > 0 order by height desc limit 1)", {'
count': Number}, function (err, rows) {
                ...
                var reject = !(rows[0].count);

                modules.blocks.count(function (err, count) {
                    ...
                    if (reject || verify || count == 1) {
                        // 428行
                        load(count);
                    } else {
                        // 其他情况，请查看源码
                        ...
};
```

408行：将调用logic/account.js文件的createTables()方法，创建与用户相关的帐号信息表，全部以“mem_”开头，主要包括“mem_accounts”，“mem_accounts2contacts”，“mem_accounts2u_contacts”，“mem_accounts2delegates”，“mem_accounts2u_delegates”，“mem_accounts2multisignatures”，“mem_accounts2u_multisignatures”，“mem_round”等7个表。这些信息与用户相关，不需要被其他节点同步。

如果是新客户端，数据库为初始创建，创世区块第一次写入，count == 1，会立刻调用闭包load()函数（428行），加载验证缺失的区块。我们先不考虑其他情况，直接看357行的load()函数，可以发现该函数先删除帐号表格（removeTables），然后重建（createTables），并通过“async.until”方法（369行）进行循环加载区块链数据，具体方法是 modules/blocks.js的loadBlocksOffset()。当 count < offset 返回 “true” 的时候，循环结束，区块链数据同步完毕，然后触发 “blockchainReady” 事件（398行）。

这里的 “Offset” 不是分页数据，而是一次加载的区块数量，这样做的好处是避免一次性加载全部区块链，导致数据请求量过大，影响计算机性能。这个值最大是 limit 的值（380行），limit 等于 “config.json” 文件设定的全局变量 loading.loadPerIteration。用户可以修改该配置文件，在启动软件时通过命令行参数“-c”选择自己的配置文件，从而实现定制。如下：

```
// config.json文件
// 132行
"loading": {
    "verifyOnLoading": false,
    "loadPerIteration": 5000
}
```

（3）验证本地区块

上面说到 modules/blocks.js 的 loadBlocksOffset() 方法才是处理加载的具体方法，也是验证的方法所在。

```
// modules/blocks.js文件
Blocks.prototype.loadBlocksOffset = function (limit, offset, verify, cb) {
    ...
    library.dbSequence.add(function (cb) {
        library.dbLite.query("SELECT " +
            ...
            async.eachSeries(blocks, function (block, cb) {
                async.series([
                    function (cb) {
                        if (block.id != genesisblock.block.id) {
                            if (verify) {
                                // 627行 追溯区块
                                if (block.previousBlock != privated.lastBlock.id) {
                                    return cb({
                                        message: "Can't verify previous block",
                                        block: block
                                    });
                                }

                                try {
                                    // 635行 验证块签名
                                    var valid = library.logic.block.verifySignature(bl
ock);
                                }
```

```
                                ...
                                if (!valid) {
                                    return cb({
                                        message: "Can't verify signature",
                                        block: block
                                    });
                                }
                                // 650行 验证块时段（Slot）
                                modules.delegates.validateBlockSlot(block, function (e
rr) {
                                ...
                    }, function (cb) {
                        // 先给交易排序，让投票或签名交易排在前面
                        ...
                        async.eachSeries(block.transactions, function (transaction, cb
) {
                            if (verify) {
                                modules.accounts.setAccountAndGet({publicKey: transact
ion.senderPublicKey}, function (err, sender) {
                                    ...
                                    if (verify && block.id != genesisblock.block.id) {
                                        // 690行 验证交易
                                        library.logic.transaction.verify(transaction,
sender, function (err) {
                                        ...

                                });
                            } else {
                                setImmediate(cb);
                            }
                        }, function (err) {
                            if (err) {
                                // 如果出现错误，要回滚
                                async.eachSeries(transactions.reverse(), function (tra
nsaction, cb) {
                                    async.series([
                                        function (cb) {
                                            modules.accounts.getAccount({publicKey: tr
ansaction.senderPublicKey}, function (err, sender) {
                                                ...
                                                modules.transactions.undo(transaction,
 block, sender, cb);
                                            });
                                        }, function (cb) {
                                            modules.transactions.undoUnconfirmed(trans
action, cb);
                                        }
                                      ...
};
```

逐个加载区块，并验证：

627行：追溯前一区块，无法追溯自然是不正确的。

635行：验证块签名，防止块内容被篡改。建议认眞阅读该行调用的签名验证方法“verifySignature()”（在“logic/block.js”文件的150行，请去源码库查看），这应该是对二进制数据（这里是区块数据）进行签名验证的典型用法。验证失败，就要终止整个循环，删除该块及其以后的块。

650行：验证块时段（Slot），防止块位置被篡改。实际上是变相验证了区块的高度及其时间戳（相关技术请看开发实践部分《关于时间戳及相关问题》的讨论）。亿书网络按照一定的周期循环（具体请参看《DPOS机制》），每一个块都可以根据其高度计算出它的出块时段，这与它的时间戳是对应的，这就锁定了区块位置，不然就是有问题。为什么要选择验证块时段，而不是简单直接的高度或时间戳？这是因为区块链有分叉的情况，相同高度存在多个块和时间戳，但相同的块时段却只能是一个。

690行：验证交易。具体流程请参考《交易》一章的相关内容。

（4）创建新区块

上面从设计的角度，正向提供了一种实现的思路。这里，我们反向阅读代码，不再思考为什么，仅仅查看是什么，体会一下读代码是多么简单的事情。注意的是，流程图需要反向查看。按照模块设计的基本原则，处理区块链的代码，都应该集中在“modules/blocks.js”文件里，所以，我们很容易就能找到创建新区块的代码“generateBlock()”方法，如下：

```
// modules/blocks.js文件
// 1126行
Blocks.prototype.generateBlock = function (keypair, timestamp, cb) {
    // 1127行 获取未确认交易，并再次验证，放入一个数组变量里备用
    var transactions = modules.transactions.getUnconfirmedTransactionList();
    var ready = [];

    async.eachSeries(transactions, function (transaction, cb) {
        ...
        ready.push(transaction);
        ...
    }, function () {
        try {
            // 1147行
            var block = library.logic.block.create({
                keypair: keypair,
                timestamp: timestamp,
                previousBlock: privated.lastBlock,
                transactions: ready
            });
        } catch (e) {
            return setImmediate(cb, e);
        }

        // 1157行
        self.processBlock(block, true, cb);
    });
};
```

该方法很简单，1127行：获取未确认交易，并再次验证，放入一个数组变量“ready”里备用。从这里的处理方法可知道，与亿书区块关联的交易并没有实现比特币那样复杂的处理方法。比特币区块链中的所有交易，以二叉树（Merkle）表示，对于存储、维护、查询、验证交易都有很多便利。亿书目前的代码没有这样的优势，将在以后的版本中优化添加。

1147行：把整理好的数据组合成块数据结构（具体形式，类似于前面的创世区块），这里因为是新建数据，重要的是keypair、timestamp、previousBlock、transactions等四个字段信息。其他字段，诸如：totalFee、reward、payloadHash等，会在“processBlock()”（1157行）执行时处理。

这里的keypair、timestamp字段是该方法的参数，需要在调用的地方传入。我们查询代码发现，仅在“modules/delegates.js”文件里调用了该方法，其方法是“loop”，很显然该方法就是产生区块的入口方法，如下：

```
// modules/delegates.js
// 492行
privated.loop = function (cb) {
    ...
    var currentSlot = slots.getSlotNumber();
    var lastBlock = modules.blocks.getLastBlock();
    ...
    // 511行
    privated.getBlockSlotData(currentSlot, lastBlock.height + 1, function (err, curren
tBlockData) {
        ...
        library.sequence.add(function (cb) {
            if (slots.getSlotNumber(currentBlockData.time) == slots.getSlotNumber()) {
                // 519行
                modules.blocks.generateBlock(currentBlockData.keypair, currentBlockDat
a.time, function (err) {
                    ...
                });
            ...
};
```

去掉一些必要的判断，其实真正调用的时候，还是非常简单的。看上面“loop”方法，真正与“modules/delegates.js”模块关联的调用，是“privated.getBlockSlotData()”方法（470行），该方法从名字上理解就是获得块时段数据，为区块提供了密钥对和时间戳。由于是私有方法，可以猜想该方法一定与受托人有关系，让我们了解一下：

```
// modules/delegates.js
// 470行
privated.getBlockSlotData = function (slot, height, cb) {
    self.generateDelegateList(height, function (err, activeDelegates) {
        ...
        for (; currentSlot < lastSlot; currentSlot += 1) {
            var delegate_pos = currentSlot % constants.delegates;

            var delegate_id = activeDelegates[delegate_pos];

            if (delegate_id && privated.keypairs[delegate_id]) {
                return cb(null, {time: slots.getSlotTime(currentSlot), keypair: privat
ed.keypairs[delegate_id]});
            }
        }
        cb(null, null);
    });
};
```

从代码可以看出，该方法先获得可以生产区块的受托人列表，然后根据当前时段信息找到激活的受托人标识，继而找到对应的密钥对，所以这个密钥对是与特定时段的特定受托人相关的。

接下来，要运行“privated.loop()”方法才可以实现新建区块的操作，所以继续搜索代码，我们很轻松找到：

```
// modules/delegates.js
// 735行
Delegates.prototype.onBlockchainReady = function () {
    privated.loaded = true;
    privated.loadMyDelegates(function nextLoop(err) {
        // 743行
        privated.loop(function () {
            setTimeout(nextLoop, 1000);
        });
        ...
```

这是 “onBlockchainReady” 事件方法，上面分析了，当本地区块链加载完毕，就会调用该方法，也就是说，当区块链加载验证完毕，就可以创建新区块了。这里设定每秒钟（744行1000毫秒）调用一次“privated.loop()”方法，但是因为slot的限制，实际运行起来是每10秒产生一个块。

我们在上面的分析中知道，在同步缺失区块操作之前，还有一个更新节点信息的过程，这么一来，本地区块链加载完毕，创建新区块要比同步缺失的区块要早。这可以最大程度上保证节点创建区块的奖励，并让节点最大程度的服务亿书网络，当然，也增加了同步区块操作的难度。

（5）产生区块链分叉

具体到编码，区块链什么时候产生分叉？显然应该在新区块写入区块链的时候，也就是modules/blocks.js文件的1157行“processBlock()”执行的时候。检索该方法的调用，发现在1174行“onReceiveBlock()”方法里，以及1084行“loadBlocksFromPeer()”方法里，都调用了该方法，只有在“loadBlocksFromPeer()”方法里，“processBlock()”方法不执行广播操作。

该方法很长，但并不复杂，下面仅仅粘贴与分叉相关的源码，如下：

```
// modules/blocks.js文件
// 800行
Blocks.prototype.processBlock = function (block, broadcast, cb) {
    ...
    if (block.previousBlock != privated.lastBlock.id) {
        // 859行 高度相同，父块不同
        modules.delegates.fork(block, 1);
        return done("Can't verify previous block: " + block.id);
    }
    //
    if (err) {
        // 877行 受托人时段不同
        modules.delegates.fork(block, 3);
        return done("Can't verify slot: " + block.id);
    }

    if (tId) {
        // 910行 交易已经存在
        modules.delegates.fork(block, 2);
        setImmediate(cb, "Transaction already exists: " + transaction.id);
    }
};
```

上面列出了3种，还有一种：

```
// modules/blocks.js文件
// 1166行
Blocks.prototype.onReceiveBlock = function (block) {
    ...
    if (block.previousBlock == privated.lastBlock.previousBlock && block.height == pri
vated.lastBlock.height && block.id != privated.lastBlock.id) {
        // 1181行 高度和父块相同，但块ID不同
        modules.delegates.fork(block, 4);
        cb("Fork");
    }
    ...
```

事实上，这里写入分叉的代码“modules.delegates.fork()”方法，很简单，仅仅是向“forks_stat”表插入对应数据而已。我们需要重点关注的是，上面罗列了4种分叉，它们的具体分叉的原因是：

1. 859行：块高度相同，父块不同。可能是父块验证出现问题；
2. 910行：交易已经存在。可能用户重复提交了交易，典型的就是“双花”问题；
3. 877行：受托人时段不同。出现时段验证错误，而块时段是与时间戳相关的，所以可能是时间处理出现了问题；
4. 1166行：高度和父块都相同，但块ID不同。这是接收来的块，高度比最新块大于1，父块是最新块时才会正常写入区块链，不然只能写入分叉。块的ID信息是对块进行sha256加

密算法得出的结果，可能获得的是不同分支上的块。

（6）同步区块链，并解决分叉

我们上面说了，当加载验证本地区块结束，程序触发了“blockchainReady”事件（modules/loader.js 398行），于是各个模块里对应的“onBlockchainReady()”方法被执行。如果，查看每个模块对应的“onBlockchainReady()”方法，我们就能很轻松地了解，接下来程序在做什么。

这里，我们要考察如何从其他节点同步区块链，自然要看看节点模块里对应的方法。我们在《一个精巧的P2P网络实现》一章，已经贴出了“onBlockchainReady()”方法的代码，这里不再重复。请看对应源码的364行，该行在更新了节点之后，调用了“library.bus.message('peerReady')”方法，触发了另一个“peerReady”事件。这才是我们最想要的，再次回到modules/loader.js文件，该文件也定义了“peerReady”事件，如下：

```
// modules/loader.js
// 492行
Loader.prototype.onPeerReady = function () {
    setImmediate(function nextLoadBlock() {
        ...
            // 499行
            privated.loadBlocks(lastBlock, cb);
        ...
    });

    setImmediate(function nextLoadUnconfirmedTransactions() {
        ...
        // 514行
        privated.loadUnconfirmedTransactions(function (err) {
        ...

    });

    setImmediate(function nextLoadSignatures() {
        ...
        // 523行
        privated.loadSignatures(function (err) {
        ...
    });
};
```

该事件方法，通过“privated.loadBlocks()”等三个方法分别同步区块（499行）、未确认交易和签名。限于篇幅，我们仅分析“privated.loadBlocks()”方法，其他两个逻辑，请自行查阅源码。

```
// modules/loader.js
// 225行
privated.loadBlocks = function (lastBlock, cb) {
    // 226行
    modules.transport.getFromRandomPeer({
        api: '/height',
        method: 'GET'
    }, function (err, data) {
        var peerStr = data && data.peer ? ip.fromLong(data.peer.ip) + ":" + data.peer.
port : 'unknown';
        ...
        if (bignum(modules.blocks.getLastBlock().height).lt(data.body.height)) {
            ...
            if (lastBlock.id != privated.genesisBlock.block.id) {
                // 259行
                privated.findUpdate(lastBlock, data.peer, cb);
            } else { // Have to load full db
                // 261行
                privated.loadFullDb(data.peer, cb);
            }
            ...
    });
};
```

226行的“transport.getFromRandomPeer()”方法，已经在《一个精巧的P2P网络实现》一章分析过，这里不再赘述。该方法通过随机选择节点，并调用这里提供的api获得远程节点的“height”数据，在确保本地区块链高度小于远程节点区块链高度的前提下，如果本地是创世区块就把远程节点整个数据库同步过来，调用“privated.loadFullDb()”方法，不然就调用“privated.findUpdate()”方法更新缺失的区块。前者很简单，属于后者的特殊情况，仅仅分析后者即可，代码如下：

```
// modules/loader.js
// 75行
privated.findUpdate = function (lastBlock, peer, cb) {
    ...
    // 80行 获得正常块
    modules.blocks.getCommonBlock(peer, lastBlock.height, function (err, commonBlock)
{
        ...
        var toRemove = lastBlock.height - commonBlock.height;

        if (toRemove > 1010) {
            // 89行 该节点的分支太长，限制从该节点同步数据（1小时）
            library.logger.log("long fork, ban 60 min", peerStr);
            modules.peer.state(peer.ip, peer.port, 0, 3600);
            return cb();
        }

        // 暂存未确认的交易
        var overTransactionList = [];
        modules.transactions.undoUnconfirmedList(function (err, unconfirmedList) {
            ...
            async.series([
                function (cb) {
                    if (commonBlock.id != lastBlock.id) {
                        // 还能反向循环
                        modules.round.directionSwap('backward', lastBlock, cb);
                    } else {
                        cb();
                    }
                },
                function (cb) {
                    // 这里处理侧链
                    library.bus.message('deleteBlocksBefore', commonBlock);
                    // 117行 删除正常块之前的块
                    modules.blocks.deleteBlocksBefore(commonBlock, cb);
                },
                ...
                function (cb) {
                    // 129行
                    modules.blocks.loadBlocksFromPeer(peer, commonBlock.id, function (
err, lastValidBlock) {
                    ...
```

这个方法相对比较复杂，80行，为什么是获得正常块（或标准块）？因为上面有4种分叉的块，都被保存在一个数据库表“blocks”里，需要通过查询把分叉的块（不正常的）过滤掉。89行，最新块与正常块之间高度差太大，说明该节点的分支太长，暂时限制从该节点同步数据，可以提高效率，也能减少出错几率。117行，把分叉的块删除掉，就解决了分叉问题。129行，这里就跟创世块处理方法相似了。

这里需要理解的是，真实的数据库存储的数据，绝对不是像前面描述的那样，是一个个标准的抽屉罗列在一起。很多情况下，正常的区块和分叉的区块都被按照时间顺序存储，是杂乱无章的，展示给用户的是经过代码过滤之后的数据。而我们定义的区块链，是经过编码实现的理想结果，这应该不难理解。

总结

本文从技术角度，描述了区块链的相关概念，并结合源码，解读了亿书区块链相关实现，这对于直观了解和学习区块链是有帮助的。特别是社区里，有些币圈网友认为压根就不应该有区块链分叉，本文的回答可能会让他们失望了。在现实世界，理想永远是遥不可及的事情，只有更好，没有最好。因此，包括区块链技术在内，只要是介入人类经济生态的应用，永远都不是单纯的技术性问题，它的更深层次的问题在于社区，以及基于人类经济生态的政治文化。

本文没有讨论区块链相关的公共Api，因为它们都是数据库读取操作，难度不大，按照以往的惯例，都留给读者自己去阅读。整体来说，这部分相对复杂，对于区块链分叉的处理，以及与区块关联的交易，还需要更多的优化，我们会在后续的版本中升级改造。

聪明的小伙伴一定看出来了，对于创建新区块、分叉等都是受托人模块（modules/blocks.js)的核心功能，受托人是DPOS机制的内容，所以要想更深层次的把握本章知识，还应该再深入一步，切实掌握亿书的共识机制。因此，请看下一篇：《**DPOS**机制》。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

亿书白皮书： http://ebookchain.org/ebookchain.pdf

亿书官网： http://ebookchain.org

亿书官方QQ群：185046161（亿书完全开源开放，欢迎各界小伙伴参与）

参考

《精通加密货币》作者Andreas问答

区块链

关于比特币硬分叉和软分叉的争议

以太坊软分叉失败，硬分叉成功

DPOS机制

前言

共识机制是分布式应用软件特有的算法机制。在中心化的软件里，再复杂的问题都可以避开使用复杂的算法逻辑（当然，如果能用算法统领，代码会更加简洁、高效），在开发设计上可以省却一定的麻烦。但在分布式软件开发中，节点间的互操作，节点行为的统一管理，没有算法理论作为支撑，根本无法实现。所以，要想开发基于分布式网络的加密货币，共识机制无法回避。

在第一个部分，专门用一篇文章《共识机制，可编程的"利益"转移规则》来介绍共识机制的作用，也对比了当前加密货币领域常用的三种共识算法原理和优越点，但是并没有对共识算法进行深入讨论。这一篇我们就从解释"拜占庭将军问题"开始，来探讨加密货币的算法问题，并通过代码学习和研究亿书共识机制的具体实现。

源码

主要源码地址：

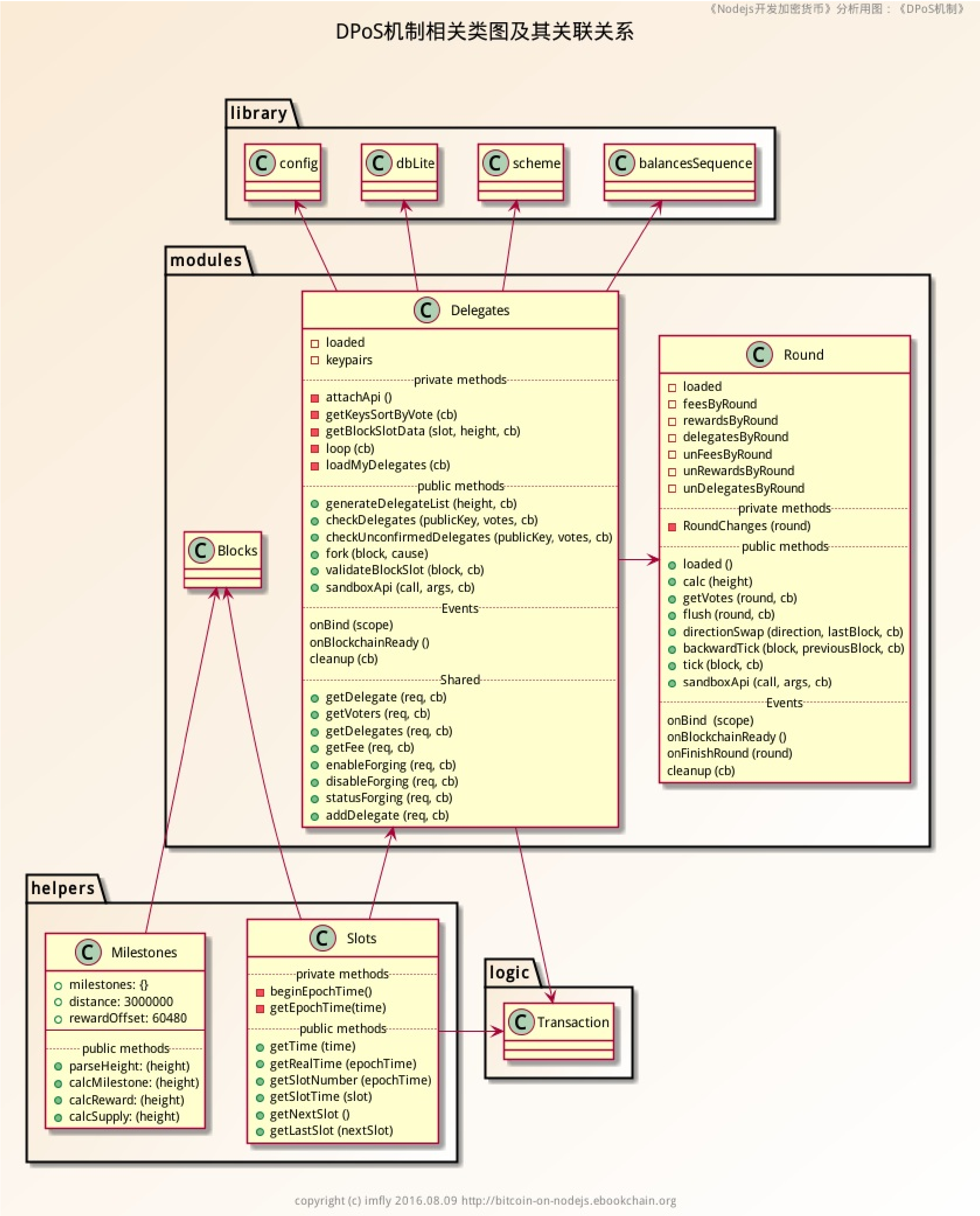
delegates.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/delegates.js

round.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/round.js

accounts.js https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/modules/accounts.js

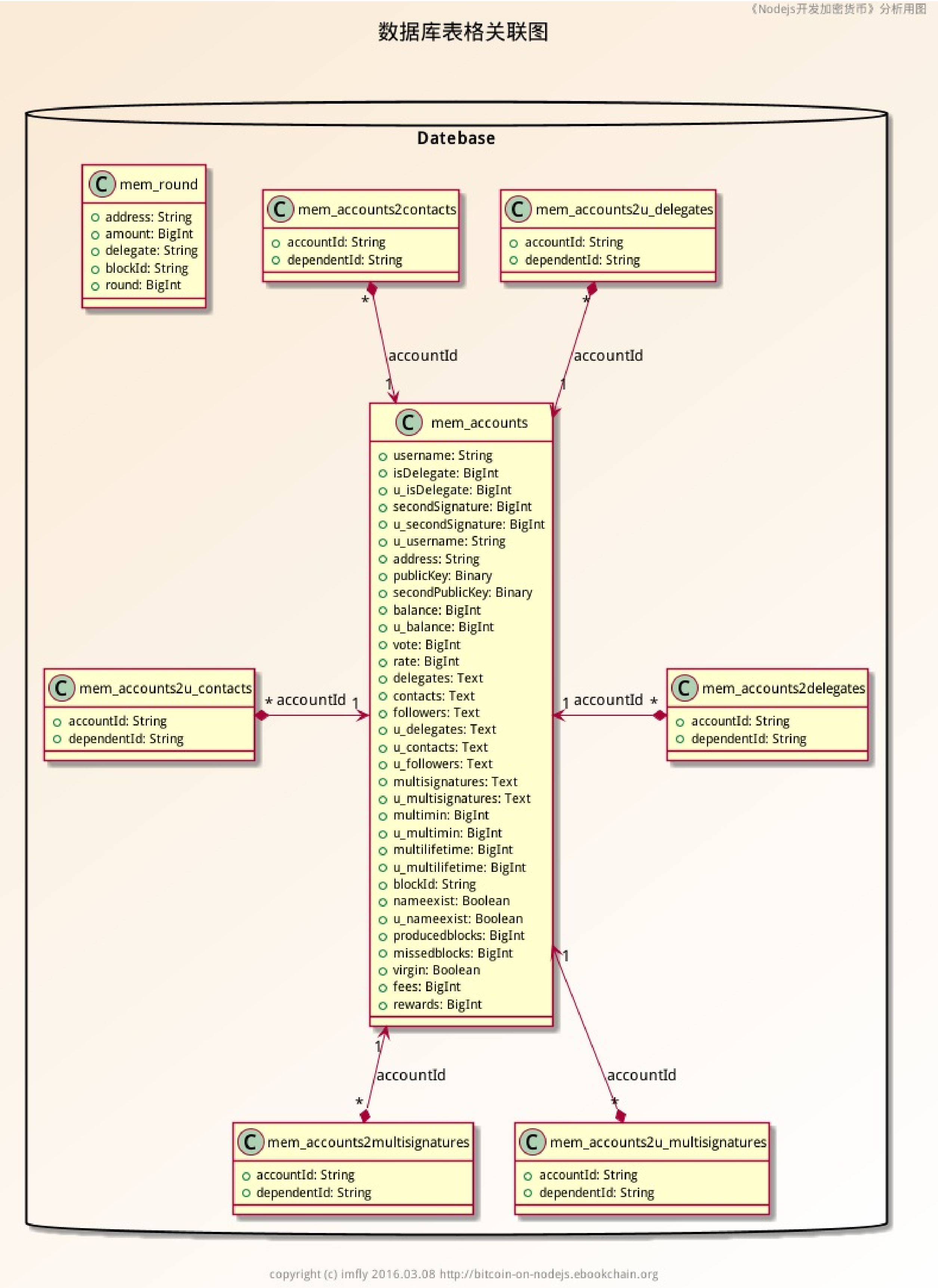
slots.js: https://github.com/Ebookcoin/ebookcoin/blob/v0.1.3/helpers/slots.js

类图



数据库表

整个数据库表应该都在该机制的管理之下，不过与用户相关的是与之直接关联的，特别是 mem_round 表。


《Nodejs开发加密货币》分析用图
数据库表格关联图
Datebase
mem_round
address: String
amount: BigInt
delegate: String
blockId: String
round: BigInt
mem_accounts2contacts
accountId: String
dependentId: String
mem_accounts2u_delegates
accountId: String
dependentId: String
accountId
accountId
mem_accounts
username: String
isDelegate: BigInt
u_isDelegate: BigInt
secondSignature: BigInt
u_secondSignature: BigInt
u_username: String
address: String
publicKey: Binary
secondPublicKey: Binary
balance: BigInt
u_balance: BigInt
vote: BigInt
rate: BigInt
delegates: Text
contacts: Text
followers: Text
u_delegates: Text
u_contacts: Text
u_followers: Text
multisignatures: Text
u_multisignatures: Text
multimin: BigInt
u_multimin: BigInt
multilifetime: BigInt
u_multilifetime: BigInt
blockId: String
nameexist: Boolean
u_nameexist: Boolean
producedblocks: BigInt
missedblocks: BigInt
virgin: Boolean
fees: BigInt
rewards: BigInt
mem_accounts2u_contacts
accountId: String
dependentId: String
accountId
accountId
mem_accounts2delegates
accountId: String
dependentId: String
accountId
accountId
mem_accounts2multisignatures
accountId: String
dependentId: String
mem_accounts2u_multisignatures
accountId: String
dependentId: String
copyright (c) imfly 2016.03.08 http://bitcoin-on-nodejs.ebookchain.org

解读

在币圈如果不知道“拜占庭将军问题”，那说明您还没有发现问题的本质。但是要想理解这个问题，如果没有点高等数学和计算机编程基础，也不是非常简单的事情。我们这里就从八卦讲起，扒扒这个概念的由来，然后对照比特币的解决办法，引出亿书机制的实现。

拜占庭将军问题

（1）比特币是怎么来的？

这里咱也八卦一下，猜猜比特币如何诞生的。在学习一门新技术的时候，我们通常会好奇，发明这项新技术的人，他是怎么想到要进行这项发明的呢？同样，对于比特币，我也曾经好奇，中本聪怎么想到要发明比特币的呢？这大量高科技的应用，可不是个小工程，一定要有明确的目的才行。这种好奇，始终督促我不断研究下去。

算法是解决问题的理论基础，拜占庭将军问题就是针对分布式共识算法提出来的，而这个问题也是比特币等加密货币的核心问题。根据比特币白皮书（更像一篇科技论文吧）内容描述，大量篇幅提到诚实节点、攻击者等问题，类似于古代战场攻防对战，很多人猜测中本聪或许就是一位专门研究这个方向的大学老师或研究人员，他解决了这个问题，推出了相关论文，并根据研究成果写出了产品原型——比特币。所以，才会说比特币仅仅是一项实验。

显然，这种猜测，纯属马后炮，由结果找原因，毫无历史根据。网上搜索了一下，各种猜测还真不少。姑且避开八卦内容，其中比较有价值的，是一篇比特金（Bit Gold）白皮书，因为它与比特币有惊人的相似之处，因此其开发者尼克·萨博（Nick Szabo）被认为最有可能是中本聪本人。比特金比比特币要早，它的目标就是实现一种不需要（只需极小的）信用中介的电子支付系统，与比特币的目标基本一致。

可见，通过使用点对点网络、加密解密等技术实现加密货币的研究由来已久，可以肯定的是，比特币的初衷绝非单纯为了解决拜占庭将军问题那么简单。相反，为了实现没有中介的电子支付系统而设计倒是更加合情合理，只不过附带完美的解决了拜占庭将军问题而已。事实上，比特币算是解决拜占庭将军问题的一个完美实现。

（2）什么是拜占庭将军问题？

但不管怎么说，拜占庭将军问题是比特币无法逾越的问题。该问题最早是由Leslie Lamport解决，为了提高宣传效果，老先生在研究论文里编造了这么一个故事，事实证明非常成功，这个故事被广泛传播。故事内容是这样的：拜占庭是东罗马帝国的首都，为了防御外敌入侵，周边驻扎了军队，而且每个军队都分隔很远，相互独立，将军与将军之间只能靠信差传递消息。在战争的时候，拜占庭军队内所有将军必需达成一致的共识（进攻或撤退），才有胜算把握。但是，在军队内部有可能存在叛徒，左右将军们的决定。这时候，在已知有成员叛变的情况下，其余忠诚的将军如何达成一致的协议，拜占庭问题就此形成。Lamport 证明了在理想状态下，背叛者为m或者更少时，将军总数只要大于3m，忠诚的将军就可以达成一致。

从技术上理解，拜占庭将军问题是分布式系统容错性问题。加密货币建立在P2P网络之上，是典型的分布式系统，类比一下，将军就是P2P网络中的节点，信使就是节点之间的通信，进攻还是撤退的决定就是需要达成的共识。如果某台独立的节点计算机拓机、掉线或攻击网络搞

破坏，整个系统就要停止运行，那这样的系统将非常脆弱，所以容许部分节点出错或搞破坏而不影响整个系统运行是必要的，这就需要算法理论上的支撑，保证分布式系统在一定量的错误节点存在的情况下，仍然保持一致性和可用性。

我非常赞同把拜占庭将军问题与两军问题分开，两个问题的本质不同，后者重在研究信差的通信问题，类似于TCP协议的握手操作，原则上是没有解的。而拜占庭将军问题是假定信差没有问题，只是将军出现了叛变等问题，所以二者本质有区别。不过，在实际的加密货币系统里，信差的问题，比如通信中断、被劫持等，都可以归为将军（节点）出了问题，理解到这一点就可以了，因此可以说比特币是完美解决了这两个问题。关于两者的区别，请看这篇文章《拜占庭将军问题深入探讨》（见附件），作者下了不少功夫，值得一读。

（3）比特币是如何解决拜占庭将军问题的？

Lamport 给出了理想状态下的答案，但现实是复杂的，比特币是如何解决的呢？事实上，比特币通过“工作量证明”（PoW）机制，简单的规范了节点（将军）的动作，从而轻松解决这个问题：

首先，维持周期循环，保证节点步调一致。这个世界上，最容易达成的就是时间上的共识，至少“几点见面”、“什么时候谈判”这样的问题很好解决吧，不然其他的都不用谈了。比特币有一个算法难度，会根据全网算力自动调整，以保证网络一直需要花费10分钟来找到一个有效的哈希值，并产生一个新区块。在这10分钟以内，网络上的参与者发送交易信息并完成交易，最后才会广播区块信息。拜占庭将军问题复杂在将军步调不一致，比特币杜绝了节点（将军们）无限制、无规律的发送命令的状态。

其次，通过算力竞赛，确保网络单点广播。将军们如果有个“带头大哥”，事情就好办了。这里的“带头大哥”可以简单的竞争得来，举个极端的例子，说好的8点钟谈判，那么先到的就是“带头大哥”，可以拟定草稿，等其他人到了签字画押就行了。“工作量证明”就是一种竞赛机制，算力好的节点，会最先完成一个新区块，在那一刻成为“带头大哥”。它把区块信息立即广播到网络，其他节点确认验证就是了。比特币通过时间戳和电子签名，实现了这样的功能，确保在某一个时间点只有一个（或几个，属于分叉行为）节点传输区块信息，改变了将军们互相传送的混乱。

最后，通过区块链，使用一个共同账本。对于单个区块，上述两条已经可以达成共识了。但现在的问题是，有一个叛徒（不诚实节点）修改了前面区块的信息，计划把钱全部划归自己所有，当它广播新区块的时候，其他节点如何通过验证？如果大家手里没有一份相同的账本，肯定无法验证，问题就会陷入僵局。基于P2P网络的BT技术是成熟的，同步一个总帐是很简单的事情。网络中的节点，在每个循环周期内都是同步的，这让每个节点（将军）做决策的时候就有了共同的基础。如果每个节点都独立维护自己的账本，问题的复杂性将无法想象，这是更广泛基础上的共识。

上述三点内容是比特币“工作量证明”（PoW）机制解决拜占庭将军问题的答案，也为其他竞争币提供了重要参考。事实上，无论你采取什么样的方式，只要保证时间统一、步调一致、单点广播、一个链条就能解决加密货币这种分布式系统的拜占庭将军问题。如果还不能深刻理

解这其中的奥妙，下面，就让我们通过阅读亿书源码，来研究DPOS（授权股权证明）机制的具体实现，去直观感受一下吧。

亿书DPOS机制概述

[亿书白皮书][] 描述了DPOS（授权股权证明）机制基本原理和改进方法，这里不再重复。亿书由受托人来创建区块，受托人来自于普通用户节点，需要首先进行注册，然后通过宣传推广，寻求社区信任并投票，获得足够排行到前101名的时候，才可以被系统接纳为真正可以处理区块的节点，并获得铸币奖励。比特币是通过计算机算力来投票，算力高的自然得票较多，容易获胜。DPOS机制是通过资产占比（股权）来投票，更多的加入了社区人的力量，人们为了自身利益的最大化会投票选择相对可靠的节点，相比更加安全和去中心化。整个机制需要完成如下过程：

（1）注册受托人，并接受投票

- 用户注册为受托人;
- 接受投票（得票数排行前101位）;

（2）维持循环，调整受托人

- 块周期：也称为时段周期（Slot），每个块需要10秒，为一个时段（Slot）；
- 受托人周期：或叫循环周期（Round），每101个区块为一个循环周期（Round）。这些块均由101个代表随机生成，每个代表生成1个块。一个完整循环周期大概需要1010秒(101x10)，约16分钟；每个周期结束，前101名的代表都要重新调整一次；
- 奖励周期：根据区块链高度，设置里程碑时间（Milestone），在某个时间点调整区块奖励。

上述循环，块周期最小（10秒钟），受托人周期其次（16分钟），奖励周期最大（347天）。

（3）循环产生新区块，广播

产生新区块和处理分叉等内容，上一章《区块链》已经讲过，这里不再赘述。

下面，我们通过源码逐个查看其实现方法。

1.注册受托人

注册受托人必须使用客户端软件（币圈俗称钱包），因此这项功能需要与节点进行交互，也就是说客户端要调用节点Api。管理受托人的模块是 modules/delegates.js ，根据前面篇章的经验，我们很容易找到该模块提供的Api：

```
"put /": "addDelegate"
```

最终的Api信息如下：

```
put /api/delegates
```

对应的方法是，modules/delegates.js模块的 `addDelegate()` 方法。该方法与注册用户别名地址等功能性交易没有区别，注册受托人也是一种交易，类型为“DELEGATE”（受托人），详细过程请自行查看modules/delegates.js文件1017行的源码，逻辑分析请参考《交易》等相关章节内容。

2.投票

我们在《交易》一章提到过，有一种交易叫 `VOTE` ，是投票交易类型。所有这类功能性交易的逻辑都很类似，就不再详细描述。这里要提示的是，该功能是普通用户具备的功能，任何普通用户都有投票权利，所以放在帐号管理模块，即“modules/accounts.js”文件里，是符合逻辑的，请参阅该文件729行的“addDelegates()”方法。当然，从代码实现上来说，放在modules/delegates.js文件里，或其他地方也都可以实现相同功能，只是逻辑上稍显混乱而已。

3.块（时段）周期（Slots）

（1）时间处理

比特币的块周期是10分钟，由工作量证明机制来智能控制，亿书的为10秒钟，仅仅是时间上的设置而已，源码在helpers/slots.js里。这个文件非常简单，时间处理统一使用UTC标准时间（请参考开发实践部分《关于时间戳及相关问题》一章），创世时间beginEpochTime()和getEpochTime(time)两个私有方法定义了首尾两个时间点，其他的方法都是基于这两个方法计算出来的时间段，所以不会出现时间上不统一的错误。

（2）编码风险

但是，唯一可能出现错误的地方，就是getEpochTime(time)方法，看下面代码的16行，new Date() 方法获得的是操作系统的时间，这个是可以人为改变的，一般情况下不会有什么影响，但个别情况也可能引起分叉行为（上一篇文章《区块链》分析过分叉的原因，其中一个就发生在这里）

```
// helpers/slots.js
function getEpochTime(time) {
    if (time === undefined) {
      // 16行
      time = (new Date()).getTime();
    }
    var d = beginEpochTime();
    var t = d.getTime();
    return Math.floor((time - t) / 1000);
}
```

（3）周期实现

从现在时间点到创世时间，有一个时间段，大小假设为 t，那么 t/10 取整，就是当前时段数（getSlotNumber()）），这里的10是由 constants.slots.interval 设定的，见文件 helpers/constants.js 25行。

具体到一个受托人，它处理的区块时段值相差应该是受托人总数，这里是101, 这个值由 constants.delegates 设定，见文件 helpers/constants.js 22行。因此，getLastSlot()方法（helpers/slots.js文件 54行）返回的是受托人最新时段值。

（4）如何使用？

块周期，是其他周期的基础，但是这里的代码并不包含任何区块、交易等关键信息。这里隐含的关联关系，就是区块、交易等信息的时间戳。只要知道任何一个时间戳，其他信息就可以使用这里的方法简单计算出来。典型的使用就是 modules/delegates.js 文件里的方法 privated.getBlockSlotData() 方法（470行），代码已在《区块链》一章贴出过，该方法为新区块提供了密钥对和时间戳。

另外，在《区块链》一章，我们也指出，程序设定每秒钟（744行1000毫秒）调用一次“privated.loop()”方法用来产生新区块，但是因为loop()方法并非每次都能成功运行，所以实际运行起来是在10秒内找到需要的受托人并产生一个区块。

（5）参数可调整吗？

很多小伙伴问，为什么一定是10秒，其他的值可以吗？为什么要101个受托人，201个可以吗？回答都是可以。我认为这个块周期如果提高到20或30或许更好（需要验证），可能更有利于SQLite数据库，也自然降低了通胀比率（后面分析）。对于计算机而言，数据库IO操作是耗时大户，10秒间隔可以提高处理交易的数量，但也会造成数据库IO无法正确完成，所以这个值是压力测试的经验值。

至于受托人的数量，也是如此。使用极限思维的方法，让受托人数量减少到1，效率高了，但是单点系统很容易受到攻击，安全性受到威胁；增加到无限大，那么查找受托人的方法将一直运行下去，结果是系统性能降到了0。所以，这个数字也是一个经验值，可以根据实际情况做适当调整。特别是那些要做DApp市场的应用，更应该考虑适当变更。

4.受托人（循环）周期（Round）

为了安全，亿书规定受托人每轮都要变更，确保那些不稳定或者做坏事的节点被及时剔除出去。另外，尽管系统会随机找寻受托人产生新块，但是在一个轮次内，每个受托人都有机会产生一个新区块（并获得奖励）并广播，这一点与比特币每个节点都要通过工作量证明机制（PoW）竞争获得广播权相比，要简化很多。

这样，亿书每个区块都会与特定的受托人关联起来，其高度(height)和产生器公钥(generatorPublicKey)必是严格对应的。块高度可以轻松找到当前块的受托人周期（modules/round.js 文件51行的calc()方法），generatorPublicKey代表的就是受托人。而单

点广播的权限也自然确定，具体代码见 modules/round.js 文件。

这里，需要重点关注的是该文件的 tick() 和 backwardTick() 方法。tick，英文意思是“滴答声或做标记”，如果大家开发过股票分析软件，在金融领域，tick还有数据快照的涵义，意思是某个时段的交易数据。我个人觉得，这里就是这个意思，是指在一个受托人周期内某个受托人的数据快照。相关数据存储在 mem_round 表里，程序退出时就会被清空。具体代码如下：

```
// modules/round.js
// 224行
Round.prototype.tick = function (block, cb) {
    ...
  // 229行
    modules.accounts.mergeAccountAndGet({
        publicKey: block.generatorPublicKey,
        producedblocks: 1,
        blockId: block.id,
        round: modules.round.calc(block.height)
    }, function (err) {
        ...
        if (round !== nextRound || block.height == 1) {
            if (privated.delegatesByRound[round].length == constants.delegates || bloc
k.height == 1 || block.height == 101) {
                var outsiders = [];
        // 255行
                async.series([
          // 256行 找到那些没在当前轮次里的节点
                    function (cb) {
                        if (block.height != 1) {
              // 258行 generateDelegateList()方法可以查询数据库里投票排行前101的节点，不过当前
可能已经改变，这里把它们找出来
                            modules.delegates.generateDelegateList(block.height, funct
ion (err, roundDelegates) {
                                ...
                                for (var i = 0; i < roundDelegates.length; i++) {
                                    if (privated.delegatesByRound[round].indexOf(round
Delegates[i]) == -1) {
                                        outsiders.push(modules.accounts.generateAddres
sByPublicKey(roundDelegates[i]));
                                    }
                                }
                                ...
                    },

          // 把缺失的块暂时保存到 mem_accounts 表里
                    function (cb) {
                        if (!outsiders.length) {
                            return cb();
                        }
                        var escaped = outsiders.map(function (item) {
                            return "'" + item + "'";
                        });
```

```
                        library.dbLite.query('update mem_accounts set missedblocks = m
issedblocks + 1 where address in (' + escaped.join(',') + ')', function (err, data) {
                            cb(err);
                        });
                    },

          // 一开始就通过merge()方法向 mem_round 表添加了记录，这里 getVotes() 方法查询出来，并
依次更新到 mem_accounts 表里。
                    function (cb) {
                        self.getVotes(round, function (err, votes) {
                            ...
                            async.eachSeries(votes, function (vote, cb) {
                                library.dbLite.query('update mem_accounts set vote = v
ote + $amount where address = $address', {
                                    address: modules.accounts.generateAddressByPublicK
ey(vote.delegate),
                                    amount: vote.amount
                                }, cb);
                ...
                        });
                    },

          // 上述变化处理完毕，接着处理余额、费用和奖励变化
                    function (cb) {
                        var roundChanges = new RoundChanges(round);

                        async.forEachOfSeries(privated.delegatesByRound[round], functi
on (delegate, index, cb) {
                            var changes = roundChanges.at(index);

                            modules.accounts.mergeAccountAndGet({
                                publicKey: delegate,
                                balance: changes.balance,
                                u_balance: changes.balance,
                                blockId: block.id,
                                round: modules.round.calc(block.height),
                                fees: changes.fees,
                                rewards: changes.rewards
                            }, function (err) {
                                ...
                    },

          // 最后，再一次更新 mem_accounts 的投票数据，没有问题就结束本轮循环（推送客户端改变），
并清除 mem_round 记录的信息
                    function (cb) {
                        self.getVotes(round, function (err, votes) {
                            ...
                            async.eachSeries(votes, function (vote, cb) {
                                library.dbLite.query('update mem_accounts set vote = v
ote + $amount where address = $address', {
                                    address: modules.accounts.generateAddressByPublicK
ey(vote.delegate),
                                    amount: vote.amount
```

```
                            }, cb);
                        }, function (err) {
                            library.bus.message('finishRound', round);
                            self.flush(round, function (err2) {
                                cb(err || err2);
                            });
                        })
                    });
                }
            ], function (err) {
                ...
```

229行，modules.accounts.mergeAccountAndGet() 方法实际上是调用 logic/account.merge() 方法（见文件logic/account.js 请自行查阅），把这里的数据插入到 “mem-”开头的数据库表里，用于处理与账户相关的一些数据，该方法比较混乱，系统需要频繁处理，对性能有较大影响，亿书将在以后的版本里做进一步优化。

255行，使用async.series方法，顺序执行了一些函数，具体情况请看代码中的注释，不再赘述。

backwardTick() 方法是反方向处理，属于回退操作，请自行查阅分析。

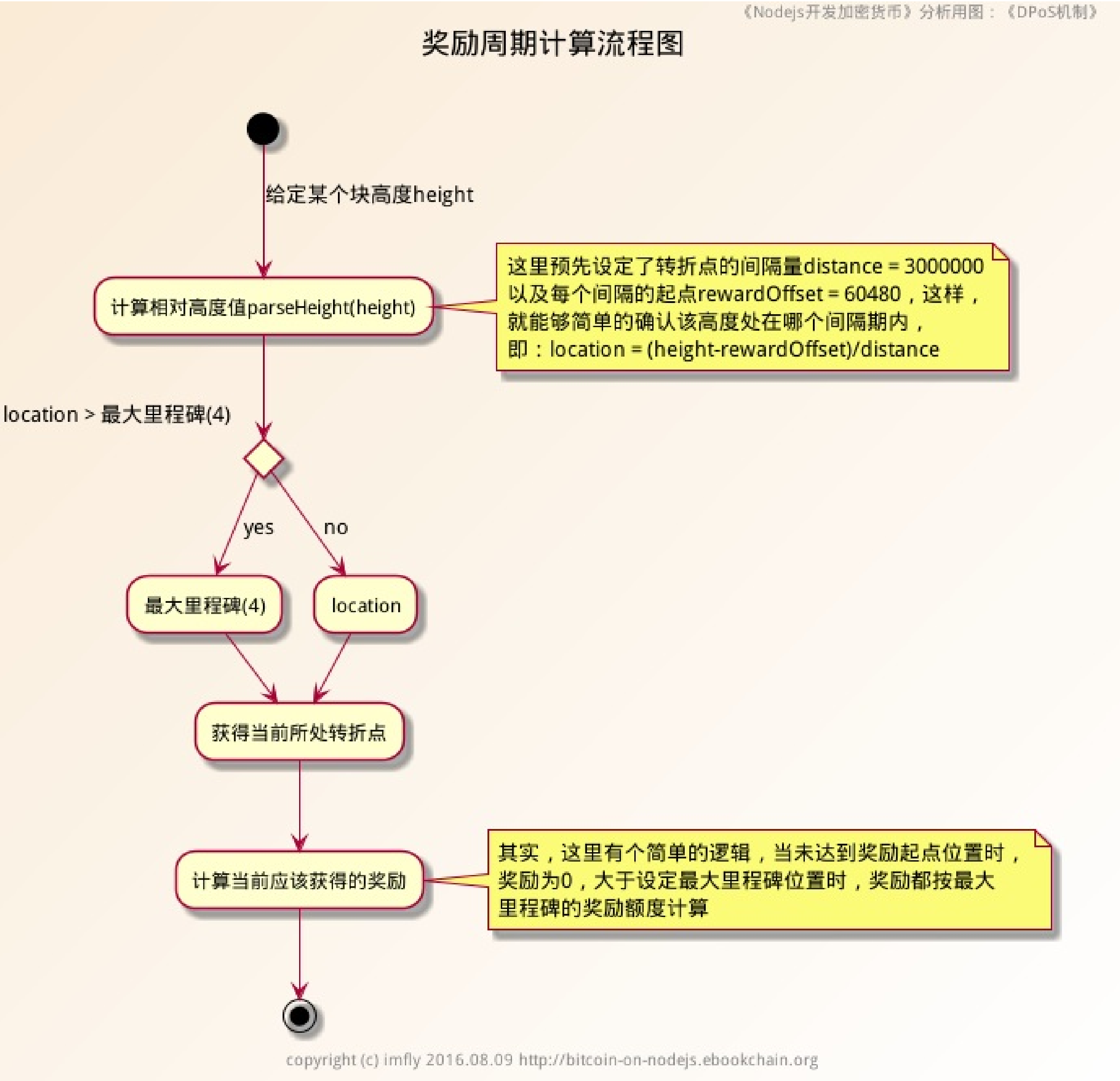
5.奖励周期（Milestones）

该周期主要针对块奖励进行设置，与比特币的块奖励每4年减半类似，亿书的块奖励也会遵循一定规则。大致的情况是这样的，第一阶段（大概1年）奖励5EBC（亿书币）/块，第二年奖励4EBC（亿书币）/块，4年之后降到1EBC(亿书币)/块，以后永远保持1EBC/块，所以总量始终在少量增发。（亿书正式上线的产品可能会做适当调整，这里仅作测试参考）

具体增发量很容易计算，第一阶段时间长度 = rewards.distance *10*秒 / （*24* 60 *60*） = *347.2* 天，增发量 = *rewards.distance* 5 = 3000000 * 5 = 1500万。第二阶段1200万，第三阶段900万，第四阶段600万，以后每阶段300万。这种适当通胀的情况是DPoS机制的一个特点，也是为了给节点提供奖励，争取更多用户为网络做贡献。

很多小伙伴担心这种通胀，会降低代币的价值，影响代币的价格。事实上，对于拥有大量侧链应用（下一篇介绍）的平台产品来说，一定要保证有足够代币供各侧链产品使用，不然会造成主链和侧链绑定紧密，互相掣肘，对整个生态系统都不是好事情。这种情况可以通过最近以太坊的运行情况体会出来，特别是侧链应用使用主链代币众筹时更不必说，此消彼长，价格波动剧烈。

具体代码见文件 helpers/milestones.js，该文件编码很简单，都是一些算术运算，请自行浏览。这里简单给出流程图：



唯一需要提醒的是，代码有一处非常隐晦的Bug，就是涉及到parseInt()方法的使用（请参考开发实践部分《Js对数据计算处理的各种问题》一章），特别是第26行。不过对系统的影响非常细微，仅仅在某个别区块高度的时候才会出现几次，比如：出现类似 parseInt(2/300000000) = 6 的情况。（亿书将在后面的版本中修改）

总结

本人介绍了拜占庭将军问题及比特币解决思路，也沿着这个思路阅读了亿书相关源码，内容涉及到多个源文件。其中，大量的细节因为篇幅所限没有详细提供，需要小伙伴们自己查阅。从源码设计上来说，这一部分的源码还需要更多的优化设计。从文章内容来说，也远远没有达到预期目的，还需要更进一步的探索和讨论。

实际上，从软件工程角度考虑，任何一个软件产品都有一个统领全局的算法机制，如果事先设计好，也会为系统开发和维护带来很大的便利，在设计的时候会有更加清晰的思路和编码方法。但是就本书而言，过多的讨论，反而偏离主线，前后繁复，所以我们暂且把更多的思考放在以后去研究。接下来，让我们把目光投向《侧链》，这也是当前极为火爆的领域。

参考

比特币白皮书：一种点对点的电子现金系统

尼克·萨博《比特金（BitGold）》白皮书

百度百科关于拜占庭将军问题描述（混淆了两军问题）

分布式共识难题（英文））

拜占庭将军问题深入探讨

第四部分：开发实践

这部分主要提供从前端到后台，亿书各产品的基础开发实践，即作为知识分享，也作为入门文档。

三张图让你全面掌握加密解密技术

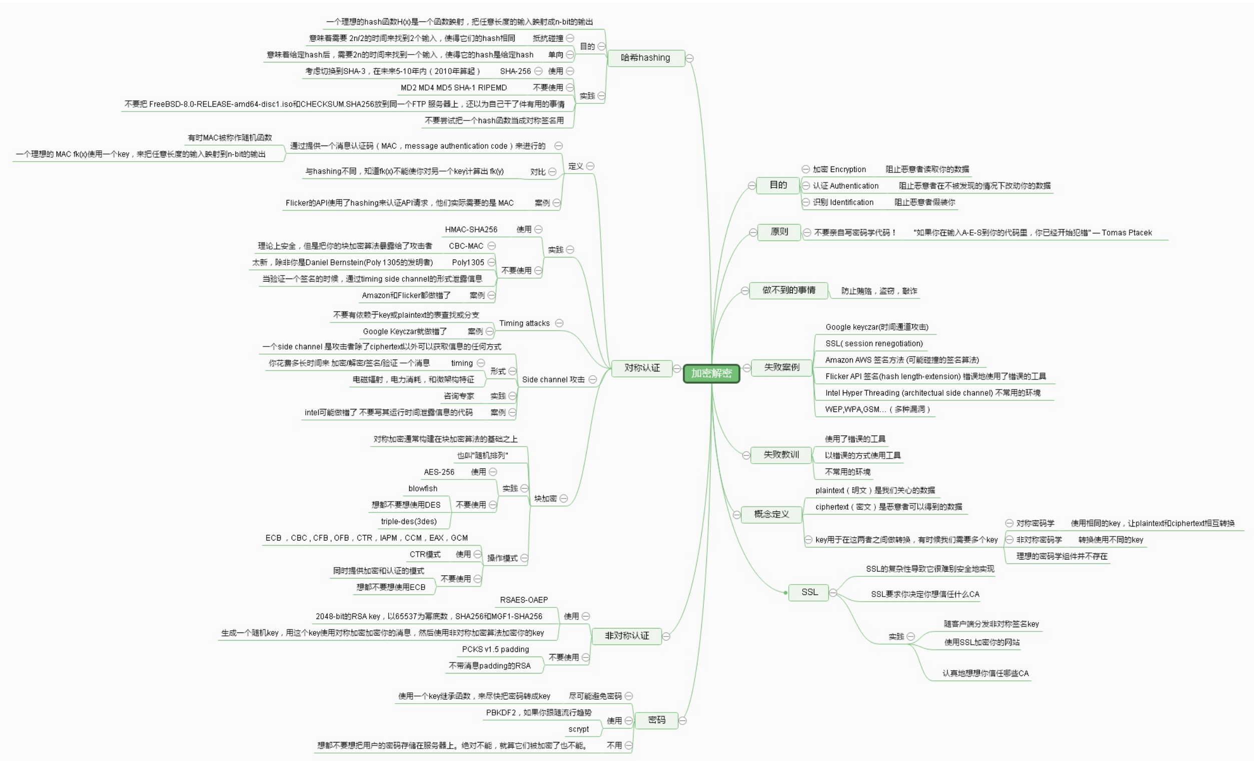
前言

加密解密技术，涉及面很广，这里，把前人的研究成果汇总起来，通过图表的形式来帮助记忆和筛选，方便日后使用。内容主要包括两个方面，一个是场景与算法，一个是Node.js的相关模块或组件。共三张脑图，具体请看：

1.加密解密纵览

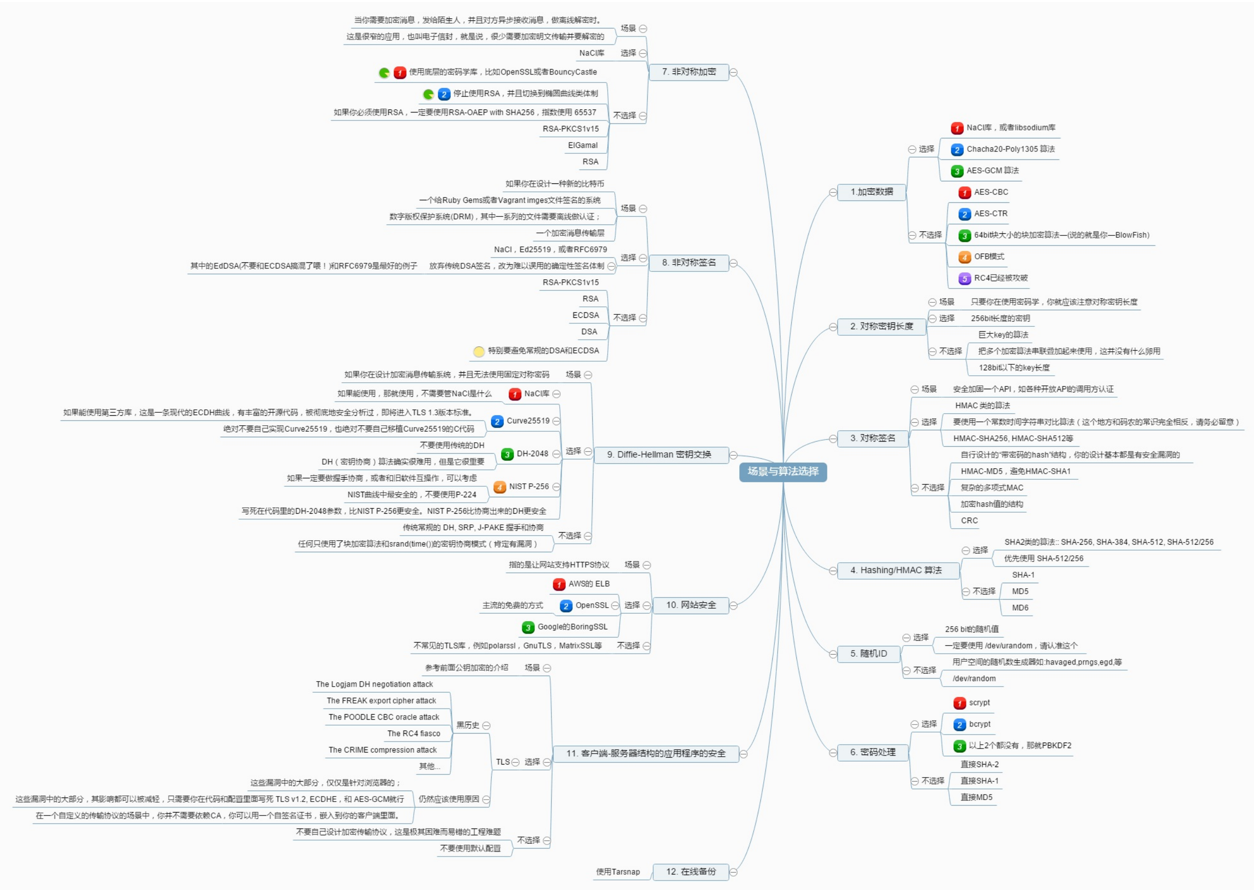
下面这张图，是在《密码学一小时必知》（见参考）基础上完成的，原作者是Colin Percival，密码学方面的专家，FreeBSD项目的安全长官，Tarsnap在线备份服务的创始人，scrypt密钥衍生算法的作者，非常值得参考学习。译者是 @byronhe ,翻译贡献这样的好文，包括下面有关论述场景与算法的实践指南，值得去为他点赞。

这张图，可以告诉你密码学中的概念，目的，案例，以及最佳的实践经验。



2.场景与算法

这张图，是基于《现代密码学实践指南(2015年)》（见参考） 完成的。可以说，在上一张图的基础上，更加具体，特别是对于场景的描述，让码农可以更加方便的作出正确的选择，值得拥有。其中，标注序号的，是有优先级的。



3.Node.js中的的加密和解密、签名与认证

这张图，主要参考了官方文档及其他一些文档（见参考），按照我个人的理解画得。如果你使用Node.js，基本上拿来看看图解，就能直接用了。特别是，默认选择了 `ed255519` 组件，如果你看了上面两篇实践，就知道这是签名与认证最好的选择，因此这里可以肯定的说，Crypto模块的签名与认证还是暂时不要用吧。 `Ebookcoin` 就是这么实践的，上一篇源码介绍的很详细。另一个是 `Natrium` 组件，也可以用于签名和认证，但主要是用来非对称加密和解密的。这张脑图里的三个组合，按照上面的实践经验来说，应该是当前Node.js加解密应用领域的最佳组合方案。

4.趣味实践

还是用在《在Node.js中使用加密解密技术》里的的例子吧。设定角色，男生叫做Bob，他的女友叫Alice。

场景

Bob想向女友表达埋藏已久的心声“I love you！”，但碍于男人的颜面（男人都这样吗？），不好意思当面说出口，只好加密传输。这里基于一个可行的假设，就是他们已经拥有彼此的公钥，或者可以简单获得。

需求

- 加密：不能让别人看到信息;
- 解密：女友可以恢复并查看;
- 签名：Bob可以签名信息，确保不被篡改;
- 认证：女友收到信息，可以验明正身，确认是Bob所发，而不是别人的恶作剧。

方案

利用以上三张图，我们可以很快拿出技术方案。

- 加密与解密技术：第二张图显示说，这种加密之后又解密原文的场景非常少见。技术上，最好使用NaCl，其次是libsodium（背后仍然是NaCl），但是搜索了一下github，Node.js社区还没有相关NaCl稳定的封装包，libsodium倒是有一个Natrium（但是，写作本文时，连安装都没有成功，有验证成功的，请告诉我一声）。因此，只能选择使用Crypto简单加密和解密。
- 签名与验证技术：当然最好的选择是 `ed25519` 了。

编码

新建一个简单的Node.js工程， 代码在这里： https://github.com/imfly/node.js-practice/blob/master/crypto/index.js

(1)生成密钥对

Bob没有使用随机字符串，而是使用一个密码，并采取SHA256算法生成密钥对，请看思维导图，有关hash的部分。

```
var crypto = require('crypto');
var ed25519 = require('ed25519');

var bobsPassword = 'This is my password, you don`t guess it!';
var hash = crypto.createHash('sha256').update(bobsPassword).digest();
var bobKeypair = ed25519.MakeKeypair(hash);
```

(2)给信息加密和签名

通常是先加密后签名。

这里使用Crypto给信息进行了简单加密，把Bob的公钥作为加密键值（但是既然是公钥，谁会不知道呢，除非Bob只把公钥给了Alice），可能还得Bob告诉Alice使用什么算法来解密。

```
var message = 'Hi Alice, I love you!';
var msgCiphered = cipher('aes192', bobKeypair.publicKey, message); //公钥进行加密，如果是Natrium，这里就是私钥加密
var signature = ed25519.Sign(new Buffer(msgCiphered, 'utf8'), bobKeypair.privateKey); //私钥进行签名
```

(3)给Alice发送签名信息

这个就各显神通了。

(4)Alice验证并解密

通常是先验证后解密。

作为Bob的好朋友，Alice有他的公钥。

```
if (ed25519.Verify(new Buffer(msgCiphered, 'utf8'), signature, bobKeypair.publicKey))
{
    // 验证函数返回了true，通过验证
  var msg = decipher('aes192', bobKeypair.publicKey, msgCiphered);  //使用Bob的公钥解密

    console.log('签名合法，信息来自Bob！');
  console.log('Bob said: ', msg); //显示信息
} else {
    // 验证函数返回了false，肯定不是Bob的信息.
    console.log('签名不合法！');
}
```

(5)补充代码

上面用到的Crypto的加密解密方法：

```
//解密
function (algorithm, key, buffer){
    var encrypted = "";
    var cip = crypto.createCipher(algorithm, key);
    encrypted += cip.update(buffer, 'utf8', 'hex');
    encrypted += cip.final('hex');
    return encrypted;
}

//解密
function decipher(algorithm, key, encrypted){
    var decrypted = "";
    var decipher = crypto.createDecipher(algorithm, key);
    decrypted += decipher.update(encrypted, 'hex', 'utf8');
    decrypted += decipher.final('utf8');
    return decrypted;
}
```

(6)运行实例

使用下面的命令，可以运行上述代码：

```
$ git clone https://github.com/imfly/node.js-practice
$ cd node.js-practice
$ npm install
$ node crypto/
```

输出结果：

```
签名合法，信息来自Bob！
Bob said:  Hi Alice, I love you!
```

链接

本系列文章即时更新，若有兴趣，可通过 `Star` 收藏，^-^

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org

电子书下载： 下载页面 PDF文件 ePub文件 Mobi文件

参考

Ed25519第三方组件

Ed25519官方网站

现代密码学实践指南(2015年)

密码学一小时必知

浅谈node.js中的Crypto模块

commander介绍

前言

从前面的源码中，我们注意到commander、domain和async三个组件（或模块）的应用，对整个app.js程序起到了至关重要的作用。作为技术积累，有必要对他们进行深入探究。这里，就按照我个人的理解，描述一下这三个组件（模块）的使用。

题外话

我坚信，搞编程的都是学习高手，智商很高。但不同的人，方法迥异。有的天生聪颖，翻翻源码和文档，就能在日后自由使用。相反，有的就差一些，我个人就算典型的一个，坚信自己愚钝至极。

我经常遇到这样的情况，对于某个组件，这次用了，下次有需要，还要再看文档，特别是在方法的选择上，有时候要反复试验。所以，多年来，我认真总结和改进自己的做法，努力向高手靠拢。

这关系到“知识”的理解和吸收。知识由语言组成，语言由概念和逻辑组成，概念和逻辑涉及到人的记忆能力与分析能力。很多人喜欢逻辑上的思考，而对于概念“视而不见”。生活中，经常见到有人描述事件的时候，用“这个、那个”等代词或形容词代替事件中的人物，应属这种现象。

事实上，概念中含有逻辑，是逻辑的高级浓缩版。概念清晰，逻辑必然是清晰的，相反却未必。因此，学习知识，掌握概念最重要。

比如，什么是模块，什么叫组件？我的理解是，Node.js提供的原生功能模块，叫`模块`（`module`，例如：domain)，而第三方提供的独立功能模块称为`组件`(`component`)，其实他们都叫`中间件`（`middleware`)，本质没什么区别。

但是，在自己的思维逻辑中，这么简单的区分一下，其实表明我们掌握了更多信息，比如在文档、代码质量、可信度等方面，`模块`天生要好于`组件`。知道`domain`是模块，自然会去node.js官网查找最可靠的资料（当然，官方计划废弃`domain`，不提倡再使用了）。

因此，我的做法就是解读概念（官方提供）、提炼概念（官方没有）、解释概念（介绍用法），用自己的理解梳理相关知识。这里的概念不以“全”为目标，以“好记、好用、好区分”为主。

我相信，以后介绍的每个组件或模块，网上都会有很多相关文档。我的做法自然也规避了文字雷同的可能，毕竟人的思维可以相似，但不会相同。

下面，以commander、domain和async在app.js中出现的顺序为准，分别介绍。

Commander

事实上，在Node.js或ruby等语言环境里，只要在文件头部添加一行所谓的`shebang`(提供一个执行环境)，就可以将代码转为命令行执行。难在命令行选项处理和流程控制，所以才有了这类工具的出现，叫它们`命令行框架`最合适。

类似`Commander`的工具有很多，但多数以规范命令行选项为主，对一些编码细节还要自己实现，比如：何时退出程序(调用`process.exit(1)`)。`Commander`把这一切都简化了，小巧灵活、简单易用，有它足够了。

1.概念定义

简单直接的命令行工具开发组件。

2.概念解释

- 这是一个`组件`，说明是第三方开发的，其实就是开发`Express`的大神`tj`开发的。ruby语言也有一个同名的开发组件，同样是`tj`的杰作，所以，虽为组件，但足够权威，“`您值得拥有`”。
- `命令行工具开发`，`Commander`的英文解释是`命令`，如其名字，这个是用来开发命令行`命令`的。
- `简单直接`，怎么简单？`四个函数`而已。怎么直接？如果您了解“命令行”的话，就能体会深刻，它通常包含命令、选项、帮助和业务逻辑四个部分，该组件分别提供了对应函数。

因此，只要记住该Commander这个名字和这一句话的概念定义，基本上已经掌握了该组件的全部。下面的用法介绍，仅仅是帮助您更好的记忆和使用。

3.用法介绍：

这里，我们也给它概念化，叫“命令行开发三步曲”。具体以 gitbook-summary为例，解释如下：

- 第1步：给工具起名字

这个`名字`，是工具的名字（其实也是`命令`，我叫它主命令），用来区分系统命令，限定`命令`使用的上下文。我通常用工程的名字或操作对象的名字代替，是个名词，比如：`book`。而用`Commander`写的`命令`是个动词（其实是用.command()方法定义的子命令），比如：`generate`，最后的形式如下：

```
$ book generate [--options]
```

只所以把起名字单独提出来，主要是在Node.js的世界里，这一步是`固定不变`的，只要记住就是了。方法是，在`package.json`里定义下面的字段：

```
{
  "bin": {
    "book": "./path/to/your-commander.js"
  }
}
```

注：`package.json`文件是包配置文件，是全局配置不可逾越之地。很多工具，都是基于它，提供入口程序的。比如：Node.js自己就是请求`main`字段的（没有定义，默认请求index.js文件)，Npm请求`scripts`字段。这里多了一个，Commander请求`bin`字段。

如果，不使用`package.json`，那么定义的就是`node`命令之下的子命令，调用方法是：

```
$ node ./path/to/your-commander.js generate [--options]
```

如果连node都不想输入，那么就要在代码第一行添加`shebang`，即：

```
#!/usr/bin/env node
```

- 第2步：填充四个函数

这一步，用于定义命令、选项、帮助和业务逻辑，完全是`Commander`概念定义的使用。其实，第三方组件，也就是起到这种`微框架`的作用。具体用法，自然最好是看 官方文档 了。这里，需要进一步思考的是，对于这个组件而言，这四个函数，最重要的是什么？

我们想到的通常是`业务逻辑`，不过，请注意，只要是开发，逻辑部分自然只能开发者自己实现，所以，`Commander`仅仅提供了一个接口函数而已。这里的`命令`，仅是一个名称。`帮助`是提示，也仅是简单的文本信息。剩下的各种`选项`，可以规范，也最为关键，才是`Commander`的可爱之处。

（1）命令： 使用`command`函数定义（子命令），例如

```
var program = require("commander");

program
    .command("summary <cmd>")
    .alias("sm") //提供一个别名
    .description("generate a `SUMMARY.md` from a folder") //描述，会显示在帮助信息里
...
```

当使用 `-h` 选项调用命令时，上述命令 `summary|sm` 会被显示在帮助信息里。这里的 `alias` 和 `description` 仅是锦上添花而已。

更复杂的，例如下面官方的例子， .command() 包含了描述信息和 .action(callback) 方法调用，就是说要用子命令各自对应的执行文件，这里就是./pm-install.js，以及 ./pm-search.js 和 ./pm-list.js等。

```
#!/usr/bin/env node

var program = require('..');

program
  .version('0.0.1')
  .command('install [name]', 'install one or more packages')
  .command('search [query]', 'search with optional query')
  .command('list', 'list packages installed')
  .command('publish', 'publish the package')
  .parse(process.argv);
```

说明：不使用 `command` 方法直接定义主命令，个人建议不要这么做。中规中矩地定义每一个子命令（本文统称命令），只要使用 `command` 方法，不带描述信息，附带 `action` 方法。如果定义类似git类型的，一连串的命令，一个一个来，显然麻烦，就把描述信息放在 `command` 里，去掉 `action` 方法，这时默认请求对应的js文件。

（2）选项：使用 `option` 方法定义，可以理解为 `命令行数据结构` 。

该函数很简单，可以方便的将文本输入转化为程序需要的数据形式。其功能如下：

- 可以设置任何数量的选项，每一个对应一个 `.option` 函数调用;
- 可以设置默认值;
- 可以提供文本、数值、数组、集合和范围等约束类型（通过提供处理函数）;
- 可以使用正则表达式;

说明： `option` 方法，基本使用就用选项名称和描述;复杂一点就要提供处理函数或默认值;再复杂就用 `arguments` 方法代替 `option` 方法，使用 `可变参数` （带 `...` 的参数）。

（3）帮助： 使用 `help` 方法输出一切有用的描述信息，这些信息通常在命令和选项的定义中，例如

```
program.help();
```

如果要定制帮助信息，就用：

```
program.on('--help',cb);
```

（4）逻辑：使用 `action` 方法注册逻辑，将代码转向执行自己的逻辑代码，当然，git类型的多命令也可以不用。例如

```
program.action(function(cmd, options) { //code });
```

- 第3步：开发业务逻辑

撰写 `action` 可以调用的代码就是了。

4.案例分析：

代码地址：https://github.com/imfly/gitbook-summary 具体内容，请自己看代码吧。

这是一个一键生成文档目录文件的命令行工具，以后可能还会加入简繁转化功能。生成的目录文件，是gitbook生成电子书的必须文件。

我从来不愿意为写作而写作，更不愿意处理任何重复性的工作，因此，隔一段时间，经常积累一堆问题清单和零零散散的文档。这个命令行工具基本上为自己定制。但发现也有很多小盆友在用，看来跟我一样，^_^。

总结

这篇仅仅介绍了 `commander` 组件的相关概念和使用，关于如何安装（简单一条命令）、如何修改权限，都没有细说。这类知识，我坚信网上一搜多的是，自行补充吧。

原本把三个组件（模块）一起整理发布的，发现写到这已经很长了，只好临时改变，单独发布了。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org

电子书下载： 下载页面 PDF文件 ePub文件 Mobi文件

参考

Node.js 命令行程序开发教程（中文）：http://www.ruanyifeng.com/blog/2015/05/command-line-with-node.html

用npm构建简单命令行（英文）：http://blog.npmjs.org/post/118810260230/building-a-simple-command-line-tool-with-npm

用Node.js开发命令行工具（英文）：http://shapeshed.com/command-line-utilities-with-node.js/

静态网站开发全景扫描

前言

在前面的入门部分，介绍了Node.js在前端开发中的应用，并通过具体项目说明了Node.js在比特币客户端领域被广泛应用。当时为了介绍Node.js入门技术，一切都是从头创建，没有引入前端框架。但在具体的项目实践中，前端是有框架可以选择的，效率和体验会有明显提升。

具体到前端框架，我的选择是Ember.js。Ember给开发带来一种飞一般的感觉，如果问前端框架哪家强，我会毫不犹豫的说 `Ember` 。（具体为什么，网上仍然争论不休，本文不做讨论）

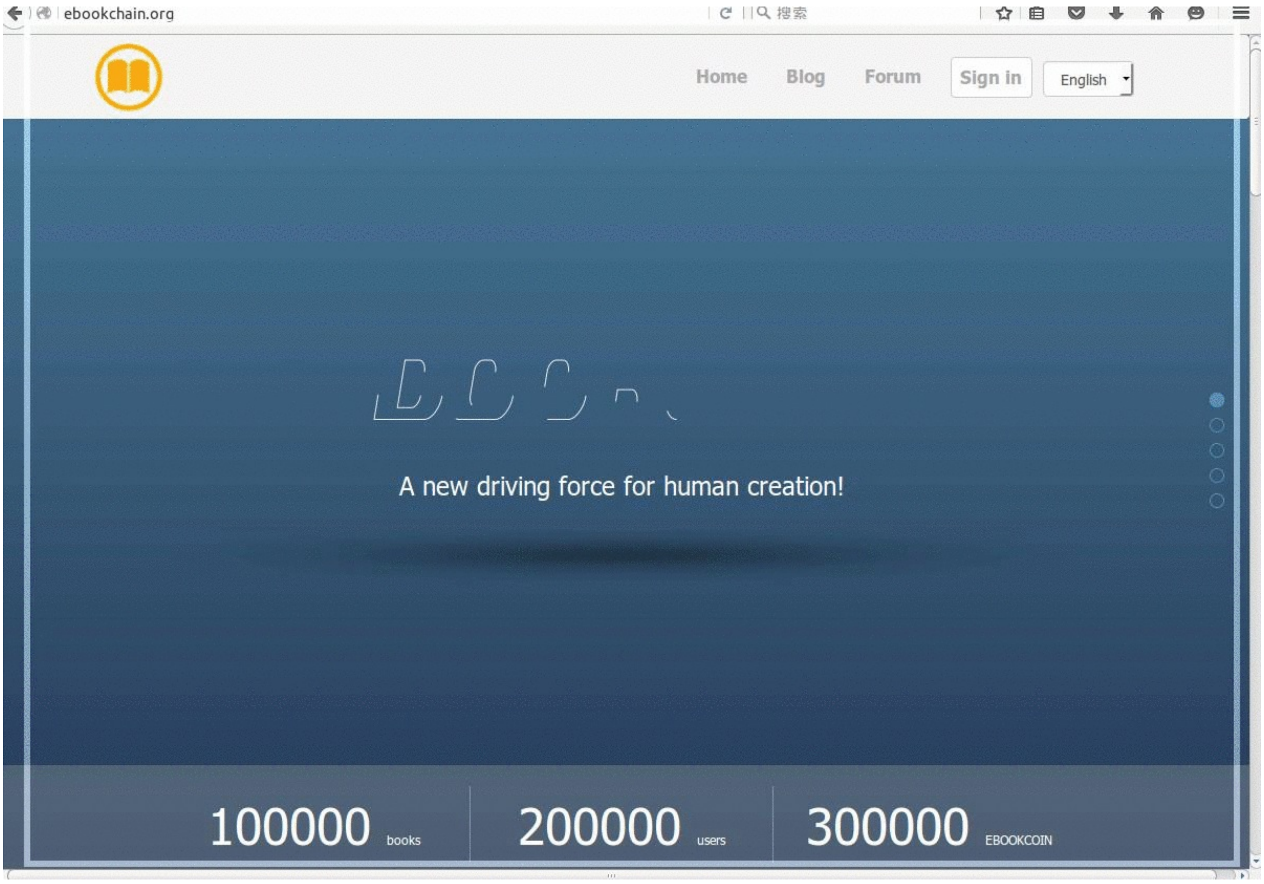
本文重点介绍静态网站的类型，亿书官网的技术选型，以及在开发亿书官网时体会到的Ember使用的几个大坑。

亿书官网介绍

1.资源信息

亿书官网： http://ebookchain.org 源码： https://github.com/Ebookchain/ebookchain.org

截图如下：



这是一个简单的静态网站，如果不考虑扩展性，单纯使用html + div + javascript的形式，对于美工较好的前端来说，就是半天的时间。

当然，对于我这样，习惯了后台开发产品的人，实际花费的时间也不多。从安装使用，搭建工程，到插件开发，引入第三方库，都统统了解了一下，所以才有了很爽快的感觉。

2.功能特点

一个静态网站还要什么特殊功能吗？好看，能传达产品信息，不就行了吗？不过，既然是研究，就得多少弄点特色出来。亿书，主要实现了以下几个功能（这些功能，很多网站都有，我把它集中到一起，因此，亿书官网可以作为一个**init**工程来用）：

- 导航动画：当滚动页面的时候，网站的header会动态调整。这个我已经抽出来，做成了Ember的插件，源码地址：https://github.com/imfly/ember-cli-animated-header
- 滚动事件：Ember没有对 `scroll` 事件的处理。这里为Ember提供了响应 `scroll` 事件的能力，抽出的插件在这里： https://github.com/imfly/ember-cli-scroller
- **SVG**动画: 当打开网站的时候，会看到第一页 `ebookchain` 的动画效果;
- 全页展示: 滚动页面，页面会按照屏幕，逐个显示出来，自动适配屏幕大小。封装的插件在这里： https://github.com/imfly/ember-cli-fullpagejs
- 多语言支持: 提供了英文和中文两种语言，默认是英文，咱也走国际范;

- 模块布局: 产品特征、合作伙伴部分（甚至footer部分）直接用的json数据，完全按照mvc模式进行分离，添加、修改、删除、扩展都很方便，无需动刀页面;
- 自动构建: 一键导出静态页面，合并压缩js,css等文件;
- 一键部署

多语言支持和扩展性，显然要比纯粹的静态页面好处多多。细心的小伙伴，一定会发现，类似的主页非常多，有的基本上完全一样。事实上，很多是直接拷贝他人的静态页面，有了亿书官网代码，建立类似的主页，扩展和修改就会简单很多。

3.技术选型

开发静态网站，可用的方案有很多，我尝试了下面三种：

（1）自己开发设计

为了延续前面的工作，最初在《Node.js开发加密货币》 入门部门提供的实例程序基础上构建了一个应用，用来输出静态页面。写到最后，发现在走ember-cli等现有产品的老路，果断放弃。（代码已经废弃删除）

（2）使用第三方产品

这类产品，有人叫做静态站点生成器，最早流行的是WordPress。不过，基于Node.js并在github上被广泛关注的，有Wintersimith，Assemble，Metalsmith，Hexo，DocPad等等。这类产品多是面向技术人员，要具备学习掌握基本安装和使用的能力。

这类产品的特点，就是帮你解决了主题、转化和部署等工作，把内容创作给你留下，极大的简化静态页面的生成过程。其原理，与我现在使用 [gitbook-summary][] 撰写电子书一样。试用了其中两款，没有简单到哪去，因此，也不是俺的菜。

（3）借助开发框架

个性化的产品，当然还得自己开发设计，只不过代码的结构和后期的处理，可以交由现成的框架产品。这样，即保证了开发设计的工作效率，也可获得更大的代码处理自由度。缺点可能是，技术门槛会高一些。

亿书选择使用Ember作为前端开发框架，涉及的产品包括：官方网站、各平台的客户端。把可以共享的部分，全部抽取出来，独立为基本的组件，方便各产品共享使用，会大大提高产品线的开发进度。试用了一下Ember，感觉很爽，不再它顾。

与Ember的前仇旧恨

之前接触过ember，那应该是1.0版本以前的事情，说起来也得有2年多了吧。当初ruby on rails火热，出来的discourse论坛就是以RoR为后台，用Ember开发的前端，很酷，真有点 `ambitions` 的感觉（Ember的宣传口号）。

不过，试用了一下，用后台的思路去开发前端应用，有点撕裂，很多东西不对路。生搬硬套后端的MVC模式，除了使开发更复杂，感觉没有太大的突破。另外，Ember是封装最为严格的前端框架，市面上大量现成的第三方开发包不能直接简单的引入使用，也成为其被受诟病的地方。

这次，为了开发 `亿书` 的官方网站，也为了给日后客户端的开发选择一个技术方案，重新看了看Ember。不看不知道，一看，我了个去，简直就是按照自己当年的期望在改进，试用了一下，岂止一个爽字了得。真心感谢，这些踏踏实实做事的团队，他们真的非常 `ambitions` 。对美丽的东西，我们得学着欣赏。

理解Ember几个让人迷乱的深“坑”

对某些小伙伴来说，ember的学习曲线还是陡了些。这里把拉几个Ember的“不良”习惯，不至于一试用就被泼了冷水。入门文章，网上有很多，就不重复了。这里提示性的，把我理解的、需要提醒小伙伴们注意的地方，简单说说。

1.什么是前端框架？

貌似高深的东西，其实也不过是一个js文件而已。因此，您完全可以像用其他js文件一样，在自己的页面里引入和使用。既然它叫框架，显然是提供了 `特定的规则` ，所以学习它的重点，就是要掌握这些规则。掌握不好，自然就会掉进“坑”里。

为什么要这么说，是因为Ember官方文档实在不是个好东西，它没有一个整体的概念，有时候让人无从知晓“为什么会如此”。即便是对细节的介绍，也不是那么细，有时候需要结合源码去理解。再者，版本、Api变化太快。这些在两年前，已经被人提出来，现在仍然没有太大改进。可见团队是多么“坚持”的一帮哥们。

2.一定要使用它的命令行工具Ember-cli

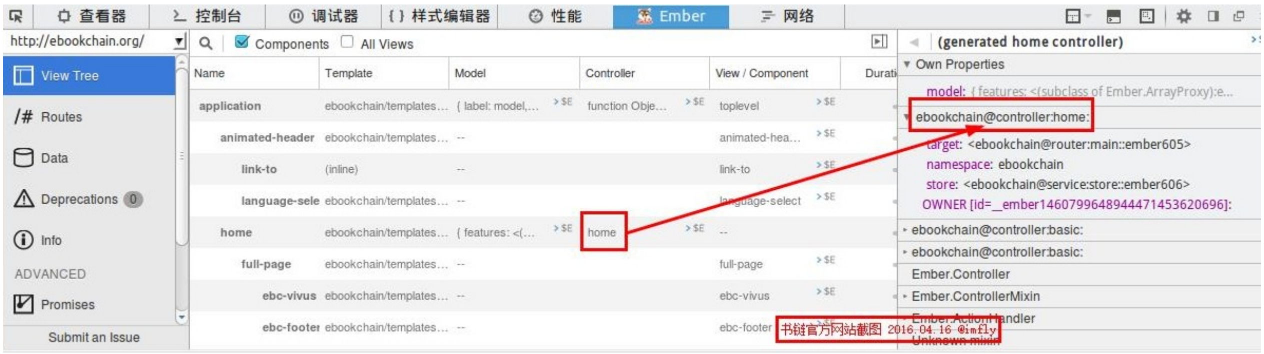
这个就别犹豫了，虽然可以直接使用js文件，但是没有Ember-cli这样强有力的命令行工具，使用Ember的难度会陡增。这个工具，让开发Ember应用，如开发后台程序，特别是用惯了ruby on rails的朋友，会非常亲切。从建立工程、产生各类逻辑代码，到测试、部署，等等，该工具（或通过插件）包揽了一切。

不习惯命令行开发？当我没说。

3.在浏览器上安装使用ember-inspector插件

这个必须有。在ruby on rails里，产生的路由等信息，可以在命令行里查看。但是，针对前端应用，只能在前端查看。通过该工具，可以掌握生成的路由、控制器、视图组件等各类对象信息，以及它们之间的对应关系，还能点击对应对象，查看对应方法和加载的数据。更主要

的是，Ember默认生成的路由和视图等也被罗列了出来，如indexRoute，如图：



4.Ember提倡的MVC模型里没有了VC

在Ember的MVC模型里，之前的版本，`M`：代表model，`V`: view（外加helper,component)，`C`：controller，路由route单独存在。2.0版本以后，这个模式改变了，VC部分逐步剔除，取而代之的是router + model + component的形式。

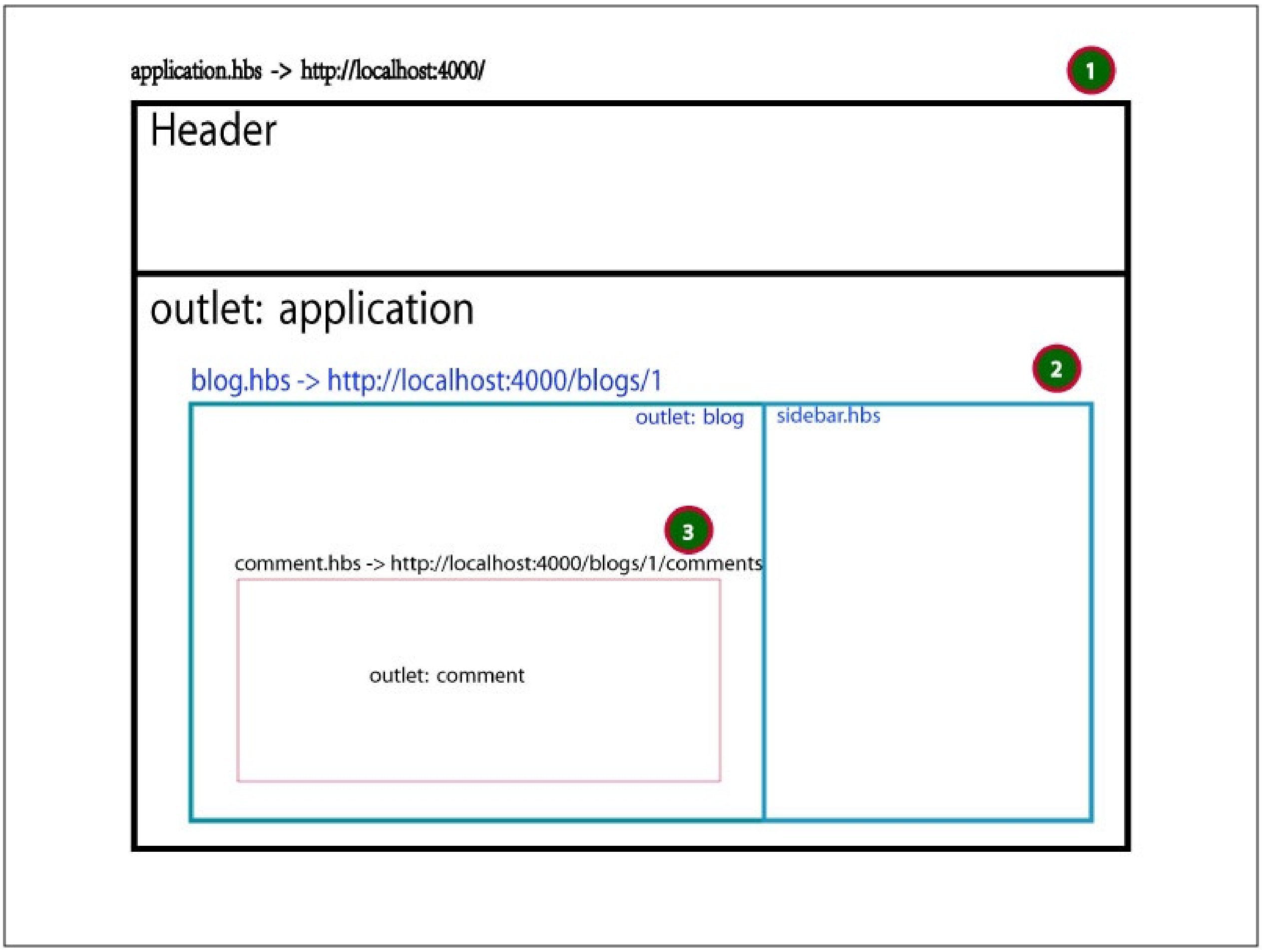
或者说，V的内容变成了component，C的内容放在了router里。个人看来，这应该是理性回归，因为之前的版本里，controller能做的事情，router也照样能做，留着controller只能是一个概念的实现，本质上没什么用处。

当然，目前的版本还保留着控制器和视图（可以通过插件来用），但是能不用就别用了。

5.有了组件，自然就没了全局模板layout和局部模板partial的存在了

理解这个很重要。一方面，Ember号称面向未来，今天开发的UI组件component，在以后也能被使用。以后的版本里，组件的地位更加重要。所以，以前可能对文章列表，比如：post-item的处理，就是写一个局部模板partial来实现重用，今天用component就好了。partial不再被支持。

另一方面，application.hbs本身就相当于一个layout.hbs文件，作为单页面应用，自然就没有所谓layout的存在，这与后台使用handlebar.js有区别。它的渲染层次如下：



这张图，权且简单的描述一下Ember的视图渲染过程，动态图像才会更直观。事实上，它还可以更复杂一些，把一些钩子方法也放进去，这样对于Ember的数据传递、视图渲染等过程就会更加直观。

6.掌握路由、模型、UI组件等各部分的钩子方法，是玩转Ember的必经之路

设计独立静止的页面，肯定没有什么难度，所以简单的hello world程序看不出什么来。现实是，多个模型的关联操作，路由状态的转移变更，UI组件的交互嵌套，插件与主程序的良好扩展，才是开发中的常态，处理它们简单了，才真叫简单。各部分之间的关联操作，多是通过不同的钩子方法实现的，请认真阅读官方文档，并在实践中总结提升。

7.学会插件开发，把一切现成的插件装进Ember里

Ember因为吸收了ruby on rails的思想，一切都是约定优于配置(convention over configuration)，所以规则最多，约束最强，这对于不习惯这些的小伙伴，是很不适应的，也导致它入门比较难（因为要学习适应的习惯比较多）。另一个就是，强约定之下，扩展插件的开发也是有一定难度的，最早很多第三方现成的js应用根本无法使用，现在已经有了较大改进，也诞生了很多优秀插件。我们下一篇，就以ember-cli-fullpagejs插件为例，单独介绍插件开发。

本地插件开发时，还有一个最容易忽视的小动作。这个不是ember的问题，但是ember的插件也是npm包，自然npm的问题也是它的问题。我们本地开发调试npm包（或ember-cli插件），通常使用 `npm link` 和 `npm link npm-name` 两个命令，将开发的npm包引入工程。最容易忽略的就是，运行完命令后，还应在工程的 `package.json` 里，添加对该包的依赖 `ember-cli-pluginName: '*'` ，不然工程是不知道的。

事实上，也有不需要设置的。不过这么做，是最稳妥的方法。建议把这一个小步骤作为一个固定 `约束` ，会节省很多时间（俺这次被坑了大半天）。

总结

这里分享的是亿书官方网站的开发体会，仅仅一个星期的体验，并不能理解它的精髓，如果让你看完感觉不是那么回事，一定不是Ember的问题，而是作者我的能力所限。Ember是个值得掌握的产品，这篇算做引子，因为使用简单，文档清晰，具体的开发使用，强烈建议浏览官方网站，这里不再详述。

写本文时，涉及到的源码还在整理中，文档和测试没有添加，功能还不健全，如果喜欢Ember，欢迎参与改进该项目。接下来两篇计划介绍它的插件开发，看看Ember是如何方便的利用现有资源的，这方面的内容官方并未提供，慢慢补全客户端开发部分的内容。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

亿书官网源码： https://github.com/Ebookchain/ebookchain.org

全屏页面插件： https://github.com/imfly/ember-cli-fullpagejs

动画效果导航：https://github.com/imfly/ember-cli-animated-header

Ember滚动事件：https://github.com/imfly/ember-cli-scroller

参考

Ember官网： https://emberjs.com

Ember-cli官网： http://ember-cli.com

亿书官网： http://ebookchain.org

开发通用的HTML组件

前言

人的懒惰常常是麻烦的开始。多数程序员都希望自己的工作一劳永逸，一次开发，到处使用，成了人人追逐的目标，我也不例外。最初写《Nodejs开发加密货币》系列文章，因为不喜欢设定好了去写，所以目录反复修改，索性弄了小工具 `gitbook-summary` ；在写入门文章的时候，反复搜索github，索性把检索与制图集成到一起，弄了个开发语言检索统计工具（见《Node.js让您的前端开发像子弹飞一样》文章实例）；阅读源码的时候，手动整理Uml图很辛苦，干脆写成了js2uml工具（见《轻松从Js文件生成UML类图》文章实例）。这里是另一个例子，不过不是辅助写作的，而是简化web开发的，希望以后用起来方便点，也是懒惰的成果之一。接下来还会有，与本书写作有关，也与亿书项目有关的一个，就是可视化部署（在部署部分介绍）。与这些小工具相比，亿书算作其中相对较大的项目了。这些工具提高了我的工作效率，但也无形中增加了很多工作量。

一个问题的解决，往往孕育着另一个问题的诞生，所以，只要写作和工作不断，与之相关的开发也就不会断。还好，除了个别情况下有点压力，我始终是享受其中的。但是，作为一个完整的有点规模的项目，明确的开发原则和开发路线图，还是必要的。其中一个重要的原则，就是保证每个功能要尽量独立，尽量做到可以重用。这不仅方便项目管理，也方便代码维护，所以，一次开发，处处可用，应该体现在每个环节。这种思想，促使我非常喜欢选择那种，稳固的、约束性较强的软件产品或开发平台，比如ruby on rails, Ember等开发框架。一旦学会，可以让我“一劳永逸”的按照一种思维逻辑去思考和解决遇到的问题。但也有聪明的小伙伴，更喜欢自己具有强大的自主控制权，这样的框架可能就不太适合他。

无论什么样的框架产品，如果一个框架，虽然强大，但是会拒很多现有的工具于门外，必然不会被大家所接受。Ember约束性较强，属于我个人爱好，最初的版本对已有开发包的兼容性较差，但是现在做了很大改进，具备很好的扩展能力，本文就结合 `ember-cli-fullpagejs` 插件的开发过程，介绍Ember-cli插件开发的各个细节，看看把一个第三方库打包成一个小小的组件是多么简单。

插件简介

样式可以浏览亿书官网，http://ebookchain.org，或参考《静态网站开发全景扫描》的截图。

（1）源码

https://github.com/imfly/ember-cli-fullpagejs

（2）使用

安装使用命令

```
$ npm install ember-cli-fullpagejs --save-dev
```

然后，只要在模板文件里，使用标签 `{{#full-page}}{{/full-page}}` 代替 `<div id="fullpage"></div>` 即可， 其他与使用 fullPage.js 一样。

必须的 **HTML** 结构

```
{{#full-page}}
    <div
    class="section">Some section</div>
    <div class="section">Some section</div>
    <div class="section">Some section</div>
    <div class="section">Some section</div>
{{/full-page}}
```

为了在一个区域里创建滑块，每个滑块默认使用包含 slide 类的元素：

```
<div class="section">
    <div class="slide"> Slide 1 </div>
    <div class="slide"> Slide 2 </div>
    <div class="slide"> Slide 3 </div>
    <div class="slide"> Slide 4 </div>
</div>
```

选项

可以给标签直接添加选项，如:

```
{{#full-page autoScrolling='true' navigation='true' anchors='["firstPage", "secondPage
"]' }}

{{/full-page}}
```

注意: 选项值必须使用单引号，而不是双引号。所有选项如下, 更多请参考:

概念解读

（1）约定优于配置(convention over configuration)

我们在《静态网站开发全景扫描》里简单罗列了Ember的几个注意点，特别提到约定优于配置的问题，这是导致很多小伙伴入手困难的根源。有人很奇怪，这是大家纷纷提倡的，本来是好事，怎么就成了问题了呢？是的，如果习惯了（其实就是记牢了）约定，开发难度会大

大降低，效率大大提高，因为框架本身已经帮你做好了这一切。相反，记不住那么多约定，或者你根本就不知道其中有这样的约定，就会给你带来很多困扰。这是目前，我们在学习很多所谓的框架知识的时候，应该特别注意的。这类框架，之所以学习成本较高，一方面是因为规则太多，另一方面就是规则与我们固有的习惯冲突太多。

举个简单的例子，我们在使用第三方库的时候，比如下面例子里的“fullPage.js”，通常要使用标签来引入，接着按照该库的逻辑去做就是了。但作为一个约束较强的前端框架，类似的工作，你要先考虑一下，是不是有了它自己的规则。事实上，在Ember框架之下，正确的使用方法是先在 `index.js` 文件里使用 `app.import` 引入文件，然后使用组件的生命周期（见参考），通过合适的钩子方法来处理，这里是 `didInsertElement()` 方法。如果仍然延续原来的做法，最好的情况是得不到任何结果，最差的情况是得出奇怪的结果。

这就给我们使用现有的第三方库造成了很大困难，原本大量现成的好工具，使用起来如此蹩脚。很多小伙伴因此，直接放弃了Ember，转投其他约束较少的框架去了。这里，我们不去衡量框架的优劣，还是直接考虑如何解决这点小问题吧。这个小例子可以帮助我们把现有的库直接改造成Ember可用的插件，让其融入Ember体系，降低绑定难度。因为插件开发的过程，与实际的开发有很多相似的地方，只不过多了一些简单的配置过程，所以我们就把具体的开发过程融入这个插件开发里一起介绍了。当然，这样做不足以介绍Ember的方方面面，至少会解决我认为最困扰我们的地方，降低Ember开发难度。

（2）浏览器世界里的组件

Ember的组件（Component）是非常重要的概念，特别是v2.0.0版本之后，全部取代了视图（View），可以理解为Ember的一切都是组件。一切都是组件的概念，大大简化了问题逻辑，也与浏览器保持了最大兼容性，甚至可以兼容未来的浏览器标准。我个人觉得，Ember团队从此终于走出了ruby on rails的桎梏，开始回归理性，真正面向前端了。毕竟把所有功能集中到一个浏览器页面里（单页面应用），还要硬生生的拉上MVC来，着实让开发者纠结不已。

我们可以把浏览器最原始的按钮、链接、下拉框等标签元素，当成Ember最基本的组件来理解。有了Ember，就可以把一篇文章、一个列表、一个图片展示区域处理成一个组件，这样做至少有三个好处：一是，开发符合MVC的要求，可以做到数据与模板分离，就像开发一个独立的页面一样，思路清晰，快速高效；二是，使用上，这个组件本身与浏览器的基础组件没有区别，非常简单直接，可以自由组合嵌套；三是，一次开发，任何地方都可使用，甚至兼容未来的浏览器。

大家看官方文档，还能看到控制器（Controller）和模型（Model）的概念，其实它们是另类的组件而已，可以理解为组件的扩展。如此以来，使用Ember就简化为浏览器组件的开发，而且使用Ember开发的组件功能也更加强大，使用与浏览器普通的组件没有分别，这样无论开发还是使用都极度简化了。如果再把今天的这个例子弄明白，基本上，我们可以把任何重复性的功能都包装成各种组件，然后打包成插件，需要的时候，直接把这些插件安装上，就可以随处可用了，就又达到了一劳永逸的效果。

开发过程

现在，我们就来看看 ember-cli-fullpagejs 的完整开发过程吧。

插件基本情况

（1）场景

Ember CLI插件API，当前支持下面的场景:

- 通过 `ember-cli-build.js` 操作 `EmberApp` （主应用）
- 添加预处理器到默认的注册表
- 提供一个自定义应用程序树与应用程序合并
- 提供定制的专用(服务)中间件
- 添加自定义模板，为主程序生成相关的工程文件

（2）命令行选项

Ember CLI有一个 *addon* 命令，带有下面的选项:

```
ember addon <addon-name> <options...>
  Creates a new folder and runs ember init in it.
  --dry-run (Default: false)
  --verbose (Default: false)
  --blueprint (Default: addon)
  --skip-npm (Default: false)
  --skip-bower (Default: false)
  --skip-git (Default: false)
```

注意：一个插件不会在已经存在的应用程序中被创建

（3）创建插件

创建一个基本插件:

`ember addon <addon-name>`

运行该命令，就会产生下面这些文件：

```
$ ember addon fullpagejs
version x.y.zz
installing
  create .bowerrc
  create .editorconfig
  create tests/dummy/.jshintrc
  ...
  create index.js

Installing packages for tooling via npm
Installed browser packages via Bower.
```

插件工程结构

通过上述命令，自动生成插件工程目录和相关文件，插件工程遵循这些结构约定:

- `app/` - 合并到应用程序的命名空间(意思是说，在使用该插件的应用程序里，可以直接使用)。
- `addon/` - 插件的命名空间部分。
- `blueprints/` - 包含插件所有蓝图模板文件，每一个存放在一个独立的文件夹里。
- `public/` - 应用程序使用的静态文件，css，images，fonts等，路径前缀 `/your-addon/*`
- `test-support/` - 合并到应用程序的 `tests/`
- `tests/` - 测试文件夹，包括一个"dummy"测试应用和验收测试助手。
- `vendor/` - 第三方专有文件，比如stylesheets, fonts, 外部包等等。
- `ember-cli-build.js` - 编译设置。
- `package.json` - Node.js元数据，依赖库等。
- `index.js` - Node.js入口(遵从npm约定)。

（1）Package.json

插件的 `package.json` 文件，像这样:

```
{
  "name": "ember-cli-fullpagejs", // 插件名称
  "version": "0.0.1", // 插件版本
  "directories": {
    "doc": "doc",
    "test": "test"
  },
  "scripts": {
    "start": "ember server",
    "build": "ember build",
    "test": "ember test"
  },
  "repository": "https://github.com/repo-user/my-addon",
  "engines": {
    "node": ">= 0.10.0"
  },
  "keywords": [
    "ember-addon"
    // 添加更多关键字，便于分类插件
  ],
  "ember-addon": {
    // 插件配置属性
    "configPath": "tests/dummy/config"
  },
  "author": "", // 你的名字
  "license": "MIT", // 协议
  "devDependencies": {
    "body-parser": "^1.2.0",
    ... // 在这里添加专门的依赖库！
  }
}
```

Ember CLI将通过检测每个应用的依赖包的 `package.json` 文件，看在 `keywords` 部分是否有 `ember-addon` 关键字，从而检查一个插件是否存在。我们还可以添加一些额外的元数据来更好地分类该插件:

```
  "keywords": [
    "ember-addon",
    "fullpagejs",
    "fullpage.js"
  ],
```

（2）插件入口

所谓的插件入口，就是调用插件最先执行的文件，每种编程语言都需要。插件将利用npm约定，并寻找一个 `index.js` 文件作为入口点，除非通过 `package.json` 文件的 `"main"` 属性指定另一个入口点。建议使用 `index.js` 作为插件入口点。

产生的 `index.js` 文件是一个简单的js对象(POJO)，可以定制和扩展，像这样：

```
// index.js
module.exports = {
  name: 'ember-cli-fullpagejs',
  included: function(app, parentAddon) {
    var target = (parentAddon || app);
    // 这里你可以修改主应用（app） / 父插件（parentAddon）. 比如, 如果你想包括
    // 一个定制的执行器，你可以把它加到目标注册器，如：
    //     target.registry.add('js', myPreprocessor);
  }
};
```

在构建（build）过程中，included钩子方法会被执行，直接操作主应用程序或者它的父插件，提高插件的处理能力。这个对象扩展了 `Addon` 类，所以任何存在于 `Addon` 类的钩子方法都可以被重写。请参考《Ember的几个重要钩子方法简介》

插件开发设计

（1）添加插件依赖

这里，我们把要封装的第三方包 `fullpagejs` 作为插件的依赖包，打包进插件里去。安装客户端依赖要通过'Bower':

```
bower install --save-dev fullpagejs
```

上述命令，自动添加bower组件到开发依赖

```
// bower.js
{
  "name": "ember-cli-fullpagejs",
  "dependencies": {
    ...
    "fullpage.js": "^2.7.8"
  }
}
```

（2）定制组件

我们要把fullpage.js定制成为一个普通的浏览器组件，希望可以这么使用它：

```
{{#full-page autoScrolling='true' navigation='true' }}

{{/full-page}}
```

先生成组件，可以使用下面的命令：

```
$ ember generate component full-page
```

组件名称至少有一个“-”线，这是约定，请记住。这个命令会自动生成必要的文件，以及测试文件，只要在里面添加逻辑代码就是了。为了允许应用程序不用手动导入语句而使用插件组件，应该把组件放在应用程序的命名空间之下，即 `app/components` 目录下，上面的命令已经帮你自动生成，如下：

```
// app/components/full-page.js

export { default } from 'ember-cli-fullpagejs/components/full-page';
```

这行代码从插件路径导入组件，再导出到应用程序。实际组件的代码放在 `addon/components/full-page.js` 里：

```
import Ember from 'ember';

export default Ember.Component.extend({
  tagName: 'div',
  // 这里的选项与fullPage.js包的选项是一致的，请参考：https://github.com/alvarotrigo/fullPag
e.js#options
  options: {
    //Navigation
    menu: '#menu',
    lockAnchors: false,
    ...
  },

  didRender() {
    Ember.run.scheduleOnce('afterRender', this, function() {
      var options = clone(this, this.options);
      Ember.$("#fullpage").fullpage(options);
    });
  },

  willDestroyElement() {
    Ember.$.fn.fullpage.destroy('all');
  }
});
```

Ember的组件渲染之后的标签，默认为 `<div></div>` ，所以如果需要改为其他的标签，比如span，可以定义tabName属性来重写。didRender()和willDestroyElement()是两个钩子方法，属于组件生命周期的一部分，前者将在组件静态内容全部渲染之后执行，起到了

$(document).ready()方法的作用，所以可以确保Ember.$("#fullpage")元素存在的时候执行；后者，将在元素销毁（通常是页面刷新的时候）的时候执行，因为Ember是一个单页面应用，无论你如何跳转或刷新，全局变量Ember.$始终保持，所以必须手动清理。

这里的问题是，为什么使用didRender()，而不是didInsertElement()钩子方法？你看看官方提供的钩子方法文档就知道了，前者在页面初始渲染以及再次渲染（刷新）的时候都可用，而后者仅在初始渲染的时候使用。也就是说，前者可以保证刷新页面，也能保证效果，后者则只能在加载页面时有效果，这也是约定好的。

这里还需要重点解释的是，这个组件也按照“约束优于配置”的原理进行了处理。比如：默认被 `{{#full-page }}` 标签包围的代码，会被包裹在 `<div id="fullpage"></div>` 里，这就避免了用户忘记设置id，而导致出现错误。另外，对选项（options）的处理，默认选项都放在了属性 `options` 之下了，但是我们却可以在使用中去重写，比如 `{{#full-page autoScrolling='true' navigation='true' }}` ，这里就使用了ember对组件的一种默认处理，即：写在组件标签里的选项，自动成为该组件对象的属性，因此 `autoScrolling` 和 `navigation` 自然就成为 `full-page` 组件的属性之一，然后我自定义clone方法，把它们重写到 `options` 就是了。看代码很简单，但却隐含了诸多知识点，也是文档没有直接提供的。

加载第三方库

（1）默认蓝图模板

所谓的蓝图模板，就是我们在安装插件的时候，应该如何把js或css文件加载到主程序，这就好比是第三方库的下载。比如本例，我们在安装插件的使用应该把fullpage.js下载到主程序里。为创建蓝图模板, 添加一个文件 `blueprints/ember-cli-fullpagejs/index.js` ，这是标准的Ember蓝图模板的命名约定。

```
module.exports = {
  description: 'ember-cli-fullpagejs',

  normalizeEntityName: function() {
    // allows us to run ember -g ember-cli-fullpagejs and not blow up
    // because ember cli normally expects the format
    // ember generate <entitiyName> <blueprint>
  },

  afterInstall: function(options) {
    return this.addBowerPackageToProject('fullpage.js', '^2.7.8');
  }
};
```

（2）加载库文件

然后，我们就可以把它导入到主应用程序了，需要在 `index.js` 文件里，使用 `included` 钩子以正确的顺序导入这些文件，如下：

```
//index.js
module.exports = {
  name: 'ember-cli-fullpagejs',

  included: function included(app) {
    this._super.included(app);

    // workaround for https://github.com/ember-cli/ember-cli/issues/3718
    if (typeof app.import !== 'function' && app.app) {
      app = app.app;
    }

    var fullpagejsPath = path.join(app.bowerDirectory, 'fullpage.js/dist');

    app.import(path.join(fullpagejsPath, 'jquery.fullpage.min.css'));
    app.import(path.join(fullpagejsPath, 'jquery.fullpage.min.css.map'), {
      destDir: 'assets'
    });

    app.import(path.join(fullpagejsPath, 'jquery.fullpage.min.js'));
    app.import(path.join(fullpagejsPath, 'jquery.fullpage.min.js.map'), {
      destDir: 'assets'
    });
  }
};
```

这一步就相当于我们平常使用的 `<script></script>` 标签，不过这里的好处是，可以直接压缩打包进主程序。

（3）导入静态文件

图片、字体等静态文件，通常放在 `/public` 文件夹里，比如有一张图片，可以保存在 `your-addon/public/images/foo.png` 路径下，使用的时候，这样调用：

```
.foo {background: url("/your-addon/images/foo.png");}
```

（4）高级定制

一般来说，如果超越内置或想要/需要更高级的控制，以下是 `index.js` 里一些插件对象的可用钩子(键)。所有的钩子都希望把一个函数作为它的值（钩子都应该是函数）。

```
includedCommands: function() {},
blueprintsPath: // return path as String
preBuild:
postBuild:
treeFor:
contentFor:
included:
postprocessTree:
serverMiddleware:
lintTree:
```

比如，这里的 `contentFor` 钩子方法，可以在主程序index.html里，含有 `{{content-for "header"}}` 标签的地方插入对应内容。

测试插件

插件工程包含一个 `/tests` 文件夹，该文件夹包含运行和设置插件测试的基本文件。`/tests` 文件夹有下面的结构:

- `/dummy`
- `/helpers`
- `/unit`
- `index.html`
- `test_helper.js`

`/dummy` 文件夹包含一个基本的dummy应用，用于测试插件。

`/helpers` 文件夹包含各类*qunit*助手，包括为了保持测试简洁，而自定义的。

`/unit` 文件夹包含单元测试，用以测试插件用于各种可用场景。

`integration/` 文件夹包含是集成测试。

`test_helper.js` 是应该在任何测试文件中引用的主要帮助文件，它导入了 `resolver` 助手，可以在 `/helpers` 文件夹中找到，用于解析 `dummy` 中的页面。

`index.html` 包含浏览器中加载的测试页面，以显示运行单元测试的结果。

对于如何设置和运行测试，请看官方文档，我们也会用一篇文章专门讲述。

蓝图模板

蓝图模板是一些具有可选安装逻辑的模板文件。它用于根据一些参数和选项生成特定的应用程序文件。一个插件可以有一个或多个蓝图模板。

（1）创建蓝图模板

给插件创建一个*blueprint*:

`ember addon <blueprint-name> --blueprint`

按照惯例，插件的主要蓝图模板应该具有与插件相同的名称:

`ember addon <addon-name> --blueprint`

在我们的例子中，使用命令:

`ember addon fullpagejs --blueprint`

这将为插件产生一个文件夹 `blueprints/ember-cli-fullpagejs` ，在这里可以定义蓝图模板的逻辑和模板文件。可以为一个插件定义多个蓝图模板。最后加载的蓝图模板会覆盖现有(同名的)蓝图模板，该模板可以是来自Ember或其他插件(根据包加载顺序)

（2）蓝图模板约定

蓝图模板应该放在插件根目录的 `blueprints` 文件夹下，就像覆盖工程根目录的蓝图模板一样。如果把它们放在插件的其他目录下，需要通过设置插件的 `blueprintsPath` 属性告诉ember-cli去哪找到它 (请看下面的 高级定制 部分)，如果熟悉 *Yeoman* (或Rails)的产生器，蓝图模板遵从类似的约定和结构。要想更深入的了解蓝图模板设计，请看 Ember CLI blueprints。

（3）模板文件结构

```
blueprints/
  fullpagejs/
    index.js
    files/
      app/
        components/
          __name__/
  unbutton
    index.js
    files/
      config/
        __name__.js
```

注：这里被命名为 `__name__` 的特殊文件或文件夹，将（在运行命令时）在主应用程序中产生一个文件/文件夹，并用第一个命令行参数(name)代替 `__name__` 。

`ember g fullpagejs my-button``

由此在主应用程序中产生一个文件夹 `app/components/my-button` 。

辅助工具

（1）开发时链接插件

当开发和测试的时候，可以在插件工程的根目录运行 `npm link` ，这样就可以通过插件名称在本地使用该插件了。然后，在计划使用的应用程序工程根目录，运行 `npm link <addon-name>` ，就会将插件链接到应用程序的 `node_modules` 文件夹下，并添加到 `package.json` 文件。这样，插件中的任何改变都会在链接该插件的任何工程中直接发生作用。

需要注意的是， `npm link` 不会像使用安装命令时那样运行默认的蓝图模板（也就是不会调用钩子方法，下载或生成相关的库文件），需要手动使用 `ember g` 来处理。另外，当我们使用这种链接的方式测试插件的时候，要提供合法的版本信息 `"<addon-name>":"version"` ，后面的version可以使用 `*` 代替，而且旧版本的npm可能需要手动添加到 `package.json` 。

（2）发布插件

使用 *npm* 和 *git* 来发布插件，就像一个标准的npm包。

```
npm version 0.0.1
git push origin master
git push origin --tags
npm publish
```

这些命令将被执行：

- 使用版本号标签版本（tag）
- 推送提交的插件代码到版本库(origin branch)
- 推送新标签到版本库(origin branch)
- 发布插件到全局npm库

（3）安装和使用插件

为了在主应用中使用插件，使用下面的命令安装该插件：

`npm install ember-cli-<your-addon-name-here> --save-dev` .

对于我们的 *fullpagejs* 插件，这样使用：

`npm install ember-cli-fullpagejs --save-dev` .

运行 *fullpagejs* 蓝图模板：

`ember generate fullpagejs`

（4）更新插件

可以像更新Ember应用一样，通过在工程根目录运行 `ember init` 命令，更新一个插件。

总结

这篇文章，通过实例详细描述了利用Ember框架，把一个第三方库封装为可以重用的组件的方法（当然，要在Ember框架之下使用），简化了第三方库的使用方法，为我们使用Ember扫除了一个障碍。但是，反过来，这篇文章可能不适合刚入门的小伙伴阅读和使用，因为大量基础知识，需要您去浏览官方文档去补充，然后结合本文，做深层次的思考。

我本人觉得，Ember的目标和代码给了我很大的触动，确实适合做比较综合的大的项目，就像亿书这类应用，大部分功能将被集中到客户端里，所以用Ember开发将非常方便。但是这不代表您也可以选择，所以做自己喜欢、擅长和有价值的事情，才是成功的开端。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

亿书白皮书： http://ebookchain.org/ebookchain.pdf

亿书官网： http://ebookchain.org

亿书官方QQ群：185046161（亿书完全开源开放，欢迎各界小伙伴参与）

区块链俱乐部公众号：chainclub

参考

- developing addons and blueprints
- 组件的生命周期
- fullPage.js
- 插件的钩子方法（hooks）文档

Ember的两个重要钩子方法介绍

`contentFor` 钩子方法

范围：这是一个插件的钩子方法。

用途：该方法可以在构建时被调用，直接在 `content-for` 标签的地方插入需要的内容。如果不是开发插件的话，就直接忽略他们就行了

描述：在默认生成的 `app/index.html` 里，有几处用到 `content-for` 标签，类似于 `{{content-for 'head'}}` 、 `{{content-for 'body'}}` ，它们需要等待某个插件的 `contentFor` 钩子方法注入内容。如：

```
// index.js
module.exports = {
  name: 'ember-cli-display-environment',

  contentFor: function(type, config) {
    if (type === 'environment') {
      return '<h1>' + config.environment + '</h1>';
    }
  }
};
```

不管 `{{content-for 'environment'}}` 在什么地方，这段代码将插入程序正在运行的当前环境。 `contentFor` 钩子方法会为每一个 `index.html` 下的 `content-for` 标签调用一次。

文档：

http://ember-cli.com/extending/#content

http://ember-cli.com/api/classes/Addon.html#method_contentFor

写入命令行

范围：插件方法。

用途：代替 `console.log` ，向命令行输出信息。

描述：每个插件都被发送给父应用的命令行输出流的实例，因此在插件的 `index.js` 里，输出信息到命令行，需要用到 `this.ui.writeLine` ，而不是 `console.log` 。例如：

```
// index.js
module.exports = {
  name: 'ember-cli-command-line-output',

  included: function(app) {
    this.ui.writeLine('Including external files!');
  }
}
```

其他钩子方法

```
includedCommands: function() {},
blueprintsPath: // return path as String
preBuild:
postBuild:
treeFor:
contentFor:
included:
postprocessTree:
serverMiddleware:
lintTree:
```

范围：插件，而且是在插件的`index.js`文件里。它的两个高级定制实例。

测试

前言

测试与开发相辅相成，一个完整的项目离不开测试，测试保证了系统的正确运行，这篇文章主要介绍nodejs下常用的测试框架mocha、should和一些基本的测试方法。

1.概念解释

单元测试：

在计算机编程中，单元测试（又称为模块测试, Unit Testing）是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。

集成测试：

整合测试又称组装测试，即对程序模块采用一次性或增殖方式组装起来，对系统的接口进行正确性检验的测试工作。整合测试一般在单元测试之后、系统测试之前进行。实践表明，有时模块虽然可以单独工作，但是并不能保证组装起来也可以同时工作。

系统测试:

系统测试是将需测试的软件，作为整个基于计算机系统的一个元素，与计算机硬件、外设、某些支持软件、数据和人员等其他系统元素及环境结合在一起测试。在实际运行(使用)环境下，对计算机系统进行一系列的组装测试和确认测试。系统测试的目的在于通过与系统的需求定义作比较，发现软件与系统定义不符合或与之矛盾的地方。

性能测试：

性能测试是对软件性能的评价。简单的说，软件性能衡量的是软件具有的响应及时度能力。因此，性能测试是采用测试手段对软件的响应及时性进行评价的一种方式。根据软件的不同类型，性能测试的侧重点也不同。

benchmarking：

基准测试是一种测量和评估软件性能指标的方法，具体做法是：在系统上运行一系列测试程序并把性能计数器的结果保存起来。这些结构称为"性能指标"。性能指标通常都保存或归档，并在系统环境的描述中进行注解。这可以让他们对系统过去和现在的性能表现进行对照比较，确认系统或环境的所有变化。

BDD：

BDD的英文全称是Behavior-Driven Development，即行为驱动开发。 它通过用自然语言书写非程序员可读的测试用例扩展了测试驱动开发方法。行为驱动开发人员使用混合了领域中统一的语言的母语语言来描述他们的代码的目的。这让开发者得以把精力集中在代码应该怎么写，而不是技术细节上，而且也最大程度的减少了将代码编写者的技术语言与商业客户、用户、利益相关者、项目管理者等的领域语言之间来回翻译的代价。

2.知识点介绍

测试框架Mocha：

mocha 是一个简单、灵活有趣的 JavaScript 测试框架，用于 Node.js 和浏览器上的 JavaScript 应用测试，大多数node开发者在使用，所以还是很可靠的。

断言库should.js：

should.js也是支持浏览器和nodejs环境，它的特色就是书写起来更像人类语言，错误提示也是表达期望-应该是什么样的（正如它的名字）。

web测试库supertest：

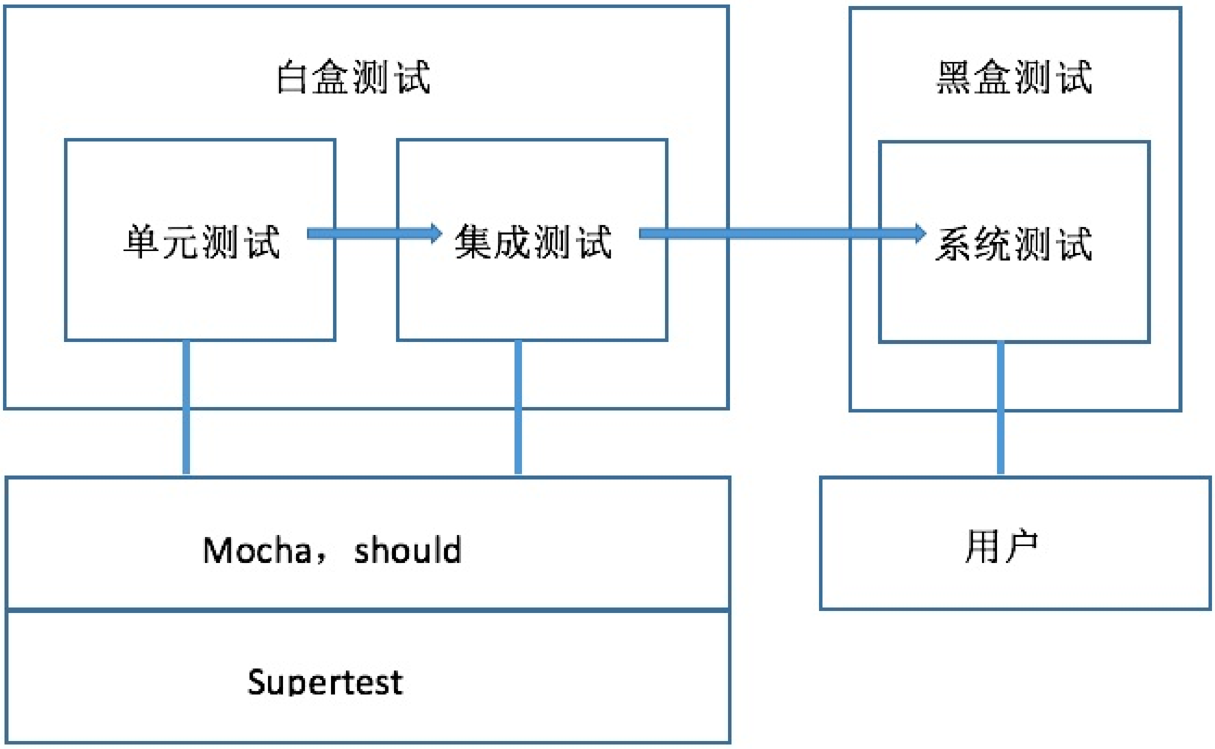
基于Super-agent的http接口测试库，提供了更高抽象的方法，使得测试http服务更加顺滑，也支持调用superagent的接口方法。

基准库benchmark:

也是一个可以用于 Node.js 和浏览器上的 JavaScript的基准测试库。

3.框架流程

我们按照测试的顺序，画出一个流程图，在不同的阶段对应了不同的测试方法。 如图：



4.实践

安装使用

```
$ npm install mocha -g
$ npm install should --save-dev
$ npm install supertest --save-dev
$ npm install benchmark --save-dev
$ mkdir test
$ atom test/test.js
```

编码

(1)简单例子

我们使用官方提供的例子，验证数组的indexOf方法越界后是否返回-1，这个例子使用了node自带的assert。

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal(-1, [1,2,3].indexOf(4));
    });
  });
});
```

- describe():传进去的字符串用来描述你要测的主体是什么，可以嵌套，这里我们要测试的主体就是Array，indexOf方法是其中的一个子描述。
- it():描述具体的 case 内容，里面包含了断言的使用。

执行后输出如下：

```
$ mocha test.js

  Array
    #indexOf()
      ✓ should return -1 when the value is not present


  1 passing (18ms)
```

上面的是passing的结果，现在我们看看failing的情况，把断言中的-1改为0，执行后可以看到：

```
$ mocha test.js

 Array
   #indexOf()
     1) should return 0 when the value is not present


 0 passing (18ms)
 1 failing

 1) Array #indexOf() should return 0 when the value is not present:

     AssertionError: 0 == -1
     + expected - actual

     -0
     +-1

     at Context.<anonymous> (test.js:5:14)
```

定位错误的信息都打印出来了，通过这个简单的例子，我们知道怎样用descride和it来描述测试用例了，接下来我们还要做一些常见的测试类型。

(2)异步回调测试
回调是js里最基础的函数调用方式，写异步测试案例我们只需要在it()添加一个回调函数(通常是done)，这个回调函数接受一个error做参数，如果传参数error就是失败，那么我们将修改刚才的代码如下，加入了延时执行：

```
var assert = require('assert');
var async = function (callback) {
  setTimeout(function () {
    callback(assert.equal(-1, [1,2,3].indexOf(4)));
  }, 10);
};
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function (done) {
    async(function (result) {
      done(result);
    });
  });
  });
});
```

这个例子输出的结果，我就不贴图了，你们可以自行试验(assert失败会返回AssertionError)。

(3)Promise测试
Promise是目前ES6规范里比较流行的异步方式了，对Promise的测试支持也变得很重要，mocha支持直接返回Promise，执行resolve是测试成功，执行reject是测试失败。

```
var assert = require('assert');
var promise = new Promise(function(resolve, reject){
  const result = [1,2,3].indexOf(4)
  if (result == 0) {
    resolve(true)
  }
  else{
    reject(false)
  }
})
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function () {
      return promise
  });
  });
});
```

(4)should.js使用

上面的例子是一些基本的判断，如果你要判断更复杂的情况，需要自己写很多代码，那么should就让这些变得简单实用，使用should库可以给对象加上更多方法，不用自己来实现判断true和false了。比如可以判断一个对象是否这个属性，判断类型是否是字符串、数组，还有很多的用法，可以去官网文档里查看http://shouldjs.github.io/#should 。

```
var assert = require('assert');
var should = require('should');

describe('Should', function() {
    it('should have property a ', function () {
      ({ a: 10 }).should.have.ownProperty('a');
  });
});
```

比较好的地方还在与should支持Promise，可以判断Promise的结果。

```
var should = require('should');

describe('Should', function() {
  it('should return 10 ', function () {
    return new Promise((resolve, reject) => resolve(10)).should.be.finally.equal(10);
  });
});
```

(5)web接口测试

此例子参照亿书的test/lib/accounts.js的代码，supertest可以直接调用superagent的接口，除此之外新增了expect接口，可以验证status、header和body，以及在end函数里直接处理返回的数据。具体接口说明可以直接查看官方文档，文章末尾会有链接。

```
var request = require('supertest');
var should = require('should');

describe('Account', function() {
    it('Opening account with password: password. Expecting success',function(done){
        request('http://127.0.0.1:7000').post('/accounts/open')
            .set('Accept', 'application/json')
            .send({
                secret: 'password'
            })
            .expect('Content-Type', /json/)
            .expect(200)
            .end(function (err, res){
                if (err) done(err);
                res.body.should.have.property('success',true);
                res.body.should.have.property('account').which.is.Object();
                res.body.account.address.should.equal('12099044743111170367C');
                res.body.account.publicKey.should.equal('fbd20d4975e53916488791477dd38
274c1b4ec23ad322a65adb171ec2ab6a0dc');
                done();
            });
    });
})
```

我们通过supertest设置请求路径和参数，对返回的response的status进行了判断，并且直接通过end函数处理返回的内容，通过刚才的should库对body的属性进行了检验，最后调用了done结束。这个过程就是在亿书中最常用的接口测试，更多例子可以去看亿书测试源码。

(6)性能测试

在上面的例子中，我们可以使用node自带的consloe.time输出耗费时间，但是如果有比较性的测试，比如需要知道哪个方案运行快，就需要benchmark库了，照着官方文档的例子，讲解一下。

```
var suite = new Benchmark.Suite;

// add tests
suite.add('RegExp#test', function() {
  /o/.test('Hello World!');
})
.add('String#indexOf', function() {
  'Hello World!'.indexOf('o') > -1;
})
// add listeners
.on('cycle', function(event) {
  console.log(String(event.target));
})
.on('complete', function() {
  console.log('Fastest is ' + this.filter('fastest').map('name'));
})
// run async
.run({ 'async': true });
```

执行的结果：

```
RegExp#test x 5,658,069 ops/sec ±1.61% (78 runs sampled)
String#indexOf x 10,170,498 ops/sec ±1.40% (78 runs sampled)
Fastest is String#indexOf
```

这个例子使用了两种方法（正则和查找索引）来判断一个字符串是否含有'o'，其中，Ops/sec 测试结果以每秒钟执行测试代码的次数（Ops/sec）显示，这个数值越大越好。除了这个结果外，同时会显示测试过程中的统计误差。

小结

这篇主要讲述了一些简单的用法和示例，测试是一门很重要的学问，测试的理论和实践有很多人做过总结，作者水平有限，难以完全论述清楚，这里只是一个简单的入门操作，让开发人员了解下nodejs的常用测试方法，如果您是专业的测试人员，或者您还需要更深入的了解，可以去看一些关于质量控制的专业书籍。

链接

本系列文章即时更新，欢迎关注，**^-^**

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org

参考

软件测试

Mocha

Should

supertest

superagent

BDD

Node.js部署方案全解析

前言

部署方案对互联网系统而言至关重要，其直接影响到系统的稳定性、可靠性、可扩展性和用户体验。部署方案涉及的范围包括很多方面，比如硬件方面有服务器性能（CPU主频、颗数、核数，内存大小）、存储方式（本地、分布式）、带宽（网络吞吐量）等，软件方面有操作系统、中间件、数据库、应用实例（进程）数目等等。对于并发量与数据可预测的系统而言，可以通过计算获得大体需求，然后针对性的选择设备或软件。而对于互联网应用这种开放式的软件系统，很容易形成并发量的爆增，从而导致系统瘫痪。除了在程序开发时要考虑高并发的处理外，还需要在部署方案上进行考虑，即要保证系统便于横向扩展，能够实现快速扩容。 在系统具备一定规模以后，部署任务将是一个繁杂的工作，任何一个环节出现问题，都极有可能导致整个系统不可用。因而部署方案应尽量降低服务间的耦合性，同时开发适合自己系统的部署与监控工具，实现自动化的部署和验证工作。部署方案的另一个重要方面就是回退方案，保证系统升级出现无法短时间解决的问题时能及时回退，并采用增量升级的方式，即一个模块一个模块、一个服务一个服务、一个集群一个集群的更新，切不可一次性全面更新，那样一旦发生问题将是灾难性的。

本质

部署方案的本质是在满足用户需求的基础上，尽量保证系统的稳定性与可用性，并最大限度的降低成本，充分利用软硬件资源。

实施

现在的部署方式基本都是使用负载均衡+分布式部署，我们也采用这样的方式。

1.硬件环境

我们目前有测试环境，预发环境，线上环境三个部分。

测试环境只有一台机器，但是上面跑了不少git分支，分别对应不同的业务，也对应不同的端口。大部分分支是“开发分支”，也就是不保证稳定，用来做开发联调的时候用的，但是在测试机器上会有一个稳定的端口（通常是8001），这个端口跑主干分支的代码，一直稳定存在，用来对接其他后台系统或者客户端的稳定分支环境。测试服务器进程用forever管理，工程师在开发的时候需要自己去服务器拉取代码，配置环境。

预发环境，是一个真实的环境，只是没有访问入口而已，在每次发布前，把代码从分支合到主干（联调前，从主干往分支合一次），然后从主干发布到预发环境，用一个指向预发环境的客户端对各个功能进行部署，测试完毕后再发布到线上环境。这个环境连接的是线上的数据库，线上的缓存，线上的依赖服务。

正式环境，考虑到初期的访问量，正式环境将由负载均衡+4台主应用服务器+一台定时任务服务器。前期考虑到成本与运维压力，我们统一采用云服务的方式。负载均衡、服务器、数据库(服务)、缓存等都是云服务。很多云服务的负载均衡已经实现了高可用和可扩展性，操作也很简单方便，解决了后期系统的横向升级需求。云应用服务器在稳定性、弹性、安全性、易用性、可扩展方面也做的很好，后期可以根据实际的并发量随时进行升级，解决了系统纵向升级的问题。

由于云服务本身都有技术团队支持，运维基本上都不需要担心了，从技术与成本的考虑，选择云服务是非常合适的。

2.软件环境

系统的软件环境主要是linux+nodejs+mysql，可以简称LNM，由于其它服务基本都是用的云，在运行时我们需要考虑的软件产品其实不多。这其实是项目前期最明智的选择。技术依赖越多，自己的特色就越少，还大量占用宝贵的人力成本，因而不是产品初期的良好方式。要想做好一个产品，应尽量把所有精力用在产品的核心上，其余的问题能交出去的就要交出去，能不做的就不要做，这一点对产品的成功与否非常重要。

至于开发环境与工具的选择，只要基础是linux+nodejs+mysql，其余的开发、测试工具完全可以按照自己的喜好来，推荐使用你最喜欢、最熟悉的工具，这样可以最大发挥个人生产力。

3.伺服工具

伺服工具主要是用来管理服务进程、监测服务运行状态、记录日志，保证系统的可用性，大体上起到嵌入式中看门狗的作用，即在系统宕掉以后能够自动重启，恢复运行并记录错误日志。

对于伺服工具的选用，简单的说，线上用PM2，测试用Forever，本地用nodemon。简单分别介绍下为什么。

关于PM2和forever，一个重而大，一个小而轻，为什么线上用pm2测试用forever？

PM2更稳定一些，forever偶尔经常会莫名其妙进程就没了，出现几率不大，但是有几率，具体原因不明。 PM2会占用端口，即使你delete，stop了一个进程，它的端口还是占着，除非你把所有的list都kill，如果你的服务器上跑了好几个服务，那就很悲剧了。线上环境还好，不会经常重启，也不会调整端口。但是测试环境就不一样了，一台机器上跑10来个node服务，端口经常还要变一变（不同分支）。用pm2简直就是个大悲剧，这时候forever反而派上了用场，一个服务对应一个进程，更灵活易控制。 nodemon 这个东西，可能很多人也都知道，用

来本地开发自动重启的，这里只是提一下，以后不需问"如何不重启进程让node代码生效"之类的问题额，重启还是要重启的，用个工具就方便多了。（一个原则，不要在node进程里保存状态，进程是可以随时被重启的）。

forever和pm2的基本用法就不详述了，文档里都有，pm2有很多强大功能，大家可以多研究下，虽然不一定用得到。

4.自动化部署

部署过程一般伴随这整个系统的始终，从开发、测试，到正式发布以及以后的运维工作都免不了了发布程序。对于一个集群式、服务式的系统而言，如果采取手动发布的方式是不可想象的。建立自己的自动部署工具基本成为必然需求。前文说过能拿来就用的尽量不要自行开发，但是一般的工具未必会完全适合你，为了满足系统自身的需求，有些工具还是自己做的比较好。

自动化部署工具其实是建立自己开发工作的机器人，对于一个开发人员而言，不运用程序为自己做些自动化的事情，实在是对不起自己的身份。个人以为，在向别人推荐自己的产品有多好之前，应该先让自己受益，那样你才能真正说服别人，让别人相信产品带来的好处。建立自己的机器人，就是要把自己从繁琐的事务中解放出来，把主要精力放在产品的提升上，放到核心工作上，让我们自己受益于我们的工作。

机器人的建立可以包含很多方面，凡是重复性、无特殊性的工作基本都可以解决。原来我工作的时候，要采购一套第三方软件，是关于图像识别的。由于要测试识别效果，经常要对几万幅图进行验证。厂家是个能偷懒的，他们把验证结果输出到一个文本文件了事，人工对照正确率真是个考验人的活。而且这件事没有任何技术含量，安排给谁都是浪费我们的宝贵人力，每次安排给一个初级测试工程师，他都要吭哧吭哧干上两天，还容易出错。后来我就写了一段代码，把测试结果统计一下，把验证图片按照结果分别复制到不同的文件夹，测试工程师只需要个文件夹一个文件夹的浏览图片，由于图片较小又归了类，一眼就可以看出很多图片的正确与否，选择一个图片，如果结果不正确按ESC，正确按Enter，一个小时基本就完成了。其实这个工作很简单，只要肯花一点时间去写点代码，可以节省几倍甚至几十倍的工作，可惜很多人宁可不断重复劳动，也不愿意去"偷懒"。

这里我们就自行实现一个自动化部署的例子。自动化部署也是属于开发机器人的一部分，怎么实现呢？我们从实际的需要分析。部署的基本过程是把服务器上的代码做好备份，本地代码（有些可能需要先编译）上传到服务器，配置成生产环境的参数，对数据库进行修改（必要时），然后重启服务。为了不影响在线用户，还要先从负载均衡设备上把要部署的服务摘下，部署完成并经过测试后，再重新挂上，然后继续下一个服务。

我们手动的操作基本就是这个流程，我们现在就把这个人工过程变成自动过程。

第一步就是打包要发布的代码，当然是你已经提交并经过测试的代码，linux打包文件就是用tar命令，nodejs调用本地命令一般是用spawn，代码如下：

```
const spawn = require('child_process').spawn;
function archive(opt, cb) {
    var src = opt.src;
    if(src.endsWith('/'))src=src.substring(0, src.length-1);
    var tmp = src.split('/');
    var dir = src.substring(0, src.lastIndexOf('/'));
    var filename = tmp[tmp.length-1];
    var archname = filename+'.tar';
    var opts = ['-cf', archname, '-C', dir, filename];
    var exclude = config.exclude

    if(exclude.length >0){
        exclude.forEach(function(exd){
            opts.push('--exclude='+exd);
        });
    }
    opts.push('--totals');

    var tar = spawn('tar', opts, { stdio: [0, 'pipe','pipe'] });

    tar.stderr.on('data', (data) => {
        console.log('error:'.red, ` ${data}`);
    });

    tar.on('close', (code) => {
        console.log('info:'.green, `package is complete.`);
        if(cb)cb();
    });
}
```

opt是配置选项，里面把源码目录，排除文件等参数设置好，即可调用 `tar` 命令进行打包。

第二步，文件上传，这里需要一个nodejs包ssh2，这个包实现了用js执行sftp等远程传输命令。我们也可以自己实现这部分功能，比如还用spawn调用scp、sftp等命令，只不过没有在服务端添加密钥的话，需要手动输入密码。ssh2可以通过程序输入密码，

```
const Client = require('ssh2');
//连接服务器选项
var options ={
    host: '192.168.*.*',//服务器地址
    port: 22,
    username: 'user',
    password: 'pass'
}
/**
*dest:目的路径，/home/web/testprj/
*arch:发布的包，/test.tar
****************************/
function upload(dest, arch){
    var conn = new Client();
    conn.on('ready', function() {
        conn.sftp(function(err, sftp) {
            if (err) throw err;
                sftp.fastPut(arch, dest+arch, function(err, result) {
                if (err) throw err;
                console.log('info:'.green, 'transport complete.');
                shutdown(conn);
            });
        });
    }).connect(options);
}
```

这样程序包即可上传服务器,接着就可以执行部署步骤了，这里还是用ssh2插件，通过ssh远程执行命令，首先进行解压：

```
function extract(conn，tar, path) {
    conn.exec('tar -xf '+tar +' -C '+path, function(err, stream) {
        if (err) throw err;
        stream.on('close', function(code, signal) {
            console.log('info: '.green, 'archive is extract successfully.');
            startup(conn);
        }).on('data', function(data) {
            console.log('info: '.green, `${data}`);
        }).stderr.on('data', function(data) {
            console.log('error: '.red, `${data}`);
        });
    });
}
```

这里conn是上面压缩时创建的ssh2的客户端连接对象，tar是上传后的包名，path是要解压到的目的路径。

之后就是具体的部署过程，比如备份，替换文件，修改配置，执行sql语句等等，都和解压命令一样，通过ssh远程调用命令执行。不同的项目具体的部署细节千差万别，可以很简单，也可以很复杂，但实现方式都没有什么本质区别，都是把手动的工作按照顺序翻译成程序执行，这里就不一一详述了。这项工作完成以后，对我们工作效率的提升是非常巨大的，也可以让枯燥的开发工作变的有趣一点。

由于部署命令大部分在控制台完成，我们有必要加入丰富的提示信息，把每一步完成的情况反馈给我们，同时为使信息清晰、整洁，我们引入colors这个npm包，它可以让控制台的字符按照我们想要的颜色显示，只需要在字符串后面加上 `.red` 等表示色彩的值就可以了。

我们还可以把这个工具写成控制台命令，类似npm这样的。这样我们可以通过命令重做部署过程的任意一步，以免部署失败后重头来过。命令行程序的开发基本就是做两件事，一是解析命令，就是获取到控制台输入的命令、选项、参数等信息，二是执行对应功能。当我们运行 `node xx.js` 时，即启动了一个node进程，这时控制台输入的所有信息都保存在 `progress.argv` 这个数组里，前两个参数就是node和 xx.js。

控制台命令的开发可以使用commander这个npm包，它提供了一个框架，你只要简单的注册命令、选项等信息，然后实现对应的功能就可以了，具体可以参考相关文档。

当然，如果你是一个系统高手，完全可以用shell命令实现相同的功能。某种程度上说，nodejs与shell有些像，他们本身都是个壳，具体做事的都是操作系统以及系统上的各个服务。

仅仅完成这些只是自动化部署的第一步，后面还要考虑怎么进行服务监测，怎么分阶段自动化部署集群等等，不过这些问题在于细节的完善，需要的是时间和精力而已。

我们可以参考一下PM2自动部署的方式，有兴趣的可以去研究下，支持多机自动部署，写个配置文件，然后执行一个命令即可，这里就不展开啦。详见文档：https://github.com/Unitech/PM2/blob/master/ADVANCED_README.md#deployment

案例

亿书官网部署方案

http://www.lxway.net/15605021.html https://certsimple.com/blog/deploy-node-on-linux Node.js as a background service

生产环境下的pm2部署

标签（空格分隔）： EBOOKCHAIN PM2 NODEJS

[TOC]

前言

部署前请先安装Ebookcoin

请参考官方wiki：https://github.com/Ebookcoin/ebookcoin/wiki/

pm2简介

Node.js默认单进程运行，对于32位系统最高可以使用512MB内存，对于64位最高可以使用1GB内存。对于多核CPU的计算机来说，这样做效率很低，因为只有一个核在运行，其他核都在闲置，pm2利用的node原生的cluster模块可以顺利解决该问题。 在线上服务器部署nodejs项目并且管理node服务会遇到诸多问题，在服务器上开启了node进程以后，我们很难统一管理进程，也不能查看详细信息和日志，如果一个接口报错了，我们需要进到服务器中，关闭进程，重启服务，然后查看命令行打印出的日志判断到底是什么问题，这明显加大了调试的难度。 pm2可以使node服务在后台运行（类似于linux的nohup），另外它可以在服务因异常或其他原因被杀掉后进行自动重启。由于Node的单线程特征，自动重启能很大程度上的提高它的健壮性。并且它是一个带有负载均衡功能的Node应用的进程管理器。当你要把你的独立代码利用全部的服务器上的所有CPU，并保证进程永远都活着，0秒的重载，PM2是完美的。它非常适合IaaS结构，但不要把它用于PaaS方案（随后将开发Paas的解决方案）。

更多信息请查看： pm2 官网：http://pm2.keymetrics.io/ pm2 github：https://github.com/Unitech/pm2

pm2主要特性

> - 原生的集群化支持（使用Node cluster集群模块）
> - 记录应用重启的次数和时间
> - 后台daemon模式运行
> - 0秒停机重载，非常适合程序升级
> - 停止不稳定的进程（避免无限循环）
> - 控制台监控
> - 实时集中log处理
> - 强健的API，包含远程控制和实时的接口API (Nodejs 模块,允许和PM2进程管理器交互)
> - 退出时自动杀死进程
> - 内置支持开机启动(支持众多linux发行版和macos）

更多特性请查看：pm2的官网：http://pm2.keymetrics.io/

pm2与forever对比

到目前为止,我们仍然依赖漂亮俏皮的node-forever模块.它是非常伟大的模块,不过依然缺失一些功能:

> - 有限的监控和日志功能
> - 进程管理配置的支持差
> - 未内置支持cluster以及优雅重启
> - 代码库老化(意味着在升级node.js时频繁的失败)
> - 未内置支持开机启动
> - 重启可能失败

pm2安装和升级

```
$ npm install -g pm2
```

pm2用法

```
基础用法：
$ pm2 start app.js -i 4  # 后台运行pm2，启动4个app.js
                         # 也可以把'max' 参数传递给 start
                         # 正确的进程数目依赖于Cpu的核心数目
$ pm2 start app.js --name my-api # 命名进程
$ pm2 list               # 显示所有进程状态
$ pm2 monit              # 监视所有进程
$ pm2 logs               # 显示所有进程日志
$ pm2 startup            # 产生init 脚本保持进程活着
$ pm2 web                # 运行健壮的 computer API endpoint (http://localhost:9615)
$ pm2 reload all/app_name       # 0秒停机重载进程，会算在服务重启的次数中，类似于平滑重启
$ pm2 restart id/all/app_name   # 会重新加载代码，因为需要杀掉原有进程，所以服务会中断
$ pm2 stop id/all/app_name      # 停止指定名称的进程，如果是一个名称多进程，则一起停止，不会释放
端口
$ pm2 delete id/all/app_name    # 删除指定名称的进程，如果是一个名称多进程，则一起删除，不会释放
端口
$ pm2 kill                      # 杀掉所有pm2进程并释放资源，包含pm2自身，会释放端口
$ pm2 updatePM2                 # 更新内存里的pm2

运行进程的不同方式：
$ pm2 start app.js -i max   # 根据有效CPU数目启动最大进程数目
$ pm2 start app.js -i 3     # 启动3个进程
$ pm2 start app.js -x       # 用fork模式启动app.js，而不是使用cluster
$ pm2 start app.js -x -- -a 23          # 用fork模式启动app.js，并且传递参数 (-a 23)
$ pm2 start app.js --name serverone     # 启动一个进程并把它命名为serverone
$ pm2 stop serverone        # 停止serverone进程
$ pm2 start app.json        # 启动进程, 在app.json里设置选项
$ pm2 start app.js -i max -- -a 23                  # 在--之后给app.js传递参数
$ pm2 start app.js -i max -e err.log -o out.log     # 启动，并将日志输出到指定文件中

你也可以执行用其他语言编写的app(fork模式)：
$ pm2 start my-bash-script.sh -x --interpreter bash         #pm2以fork模式运行bash shell
脚本
$ pm2 start my-python-script.py -x --interpreter python     #pm2以fork模式运行python脚本

硬重启：
$ pm2 dump
$ pm2 resurrect
```

pm2高级用法（远程部署）

生成远程部署所需的json文件,在应用程序目录下生成ecosystem.json

```
$ pm2 ecosystem
```

编辑修改ecosystem.json文件

```
{
    "apps" : [{
        "name" : "ebook",           //项目的名字
        "script" : "app.js",        //项目主入口（Node.js）
        "env": {
            "COMMON_VARIABLE": "true"
        },
        "env_production" : {
            "NODE_ENV": "production"
        }
    }],
deploy : {
  production : {
    user : "root",                                                  // 登陆用户名
    host : "192.168.1.100",                                         // 要部署的目标
服务器IP地址或域名
    ref  : "origin/master",                                         // 用于部署的Git
仓库分支
    repo : "https://github.com/Ebookcoin/ebookcoin.git",            // Git仓库位置
    path : "/var/www/production",                                   // 部署目标服务
器文件系统位置
    "post-deploy" : "npm install && pm2 startOrRestart ecosystem.json --env production
"     // 部署后执行的命令，本案例：先安装依赖再启动
  },
}
```

执行部署(自动发布网站项目)

```
$ pm2 deploy ecosystem.json production
```

更新部署

```
$ pm2 deploy production update
```

pm2启动模式分类

fork模式

pm2默认用fork模式启动，该模式有如下特性。

> - 可以修改exec_interpreter，比如你的代码不是纯js，而是类似coffee script或者是其它语言如python/php
> - 适合做本地开发测试用，只使用一个cpu

cluster模式

> - 不可以修改exec_interpreter，只适合node开发的纯js程序
> - 适合生产环境部署用，尽可能多地使用cpu/内存等系统资源
> - 用cluster来做负载均衡，业务代码零改动

pm2部署ebookcoin案例

本案例演示了如何利用pm2进行ebook blockchain的基本管理工作。

用pm2启动ebookcoin的app.js并将应用名字设置为“ebook”

```
clark@E540:/data/blockchain/ebookcoin$ pm2 start app.js --name ebook
[PM2] Spawning PM2 daemon with pm2_home=/home/clark/.pm2
[PM2] PM2 Successfully daemonized
[PM2] Starting /data/blockchain/ebookcoin/app.js in fork_mode (1 instance)
[PM2] Done.
┌──────────┬────┬──────┬───────┬────────┬─────────┬────────┬─────┬───────────┬────────
──┐
│ App name │ id │ mode │ pid   │ status │ restart │ uptime │ cpu │ mem       │ watchin
g │
├──────────┼────┼──────┼───────┼────────┼─────────┼────────┼─────┼───────────┼────────
──┤
│ ebook    │ 0  │ fork │ 20130 │ online │ 0       │ 0s     │ 49% │ 13.2 MB   │ disable
d │
└──────────┴────┴──────┴───────┴────────┴─────────┴────────┴─────┴───────────┴────────
──┘
 Use `pm2 show <id|name>` to get more details about an app
```

检查日志

该日志包含pm2自身、应用程序的日志（支持实时刷新，类似linxu的tail -f模式）

```
clark@E540:/data/blockchain/ebookcoin$ pm2 logs
[TAILING] Tailing last 10 lines for [all] processes (change the value with --lines opt
ion)
/home/clark/.pm2/pm2.log last 10 lines:
PM2        | 2016-09-26 15:00:25: PM2 PID file         : /home/clark/.pm2/pm2.pid
PM2        | 2016-09-26 15:00:25: RPC socket file      : /home/clark/.pm2/rpc.sock
PM2        | 2016-09-26 15:00:25: BUS socket file      : /home/clark/.pm2/pub.sock
PM2        | 2016-09-26 15:00:25: Application log path : /home/clark/.pm2/logs
PM2        | 2016-09-26 15:00:25: Process dump file    : /home/clark/.pm2/dump.pm2
PM2        | 2016-09-26 15:00:25: Concurrent actions   : 1
PM2        | 2016-09-26 15:00:25: SIGTERM timeout      : 1600
PM2        | 2016-09-26 15:00:25: ====================================================
===========================
PM2        | 2016-09-26 15:00:25: Starting execution sequence in -fork mode- for app n
ame:ebook id:0
PM2        | 2016-09-26 15:00:25: App name:ebook id:0 online

/home/clark/.pm2/logs/ebook-error-0.log last 10 lines:
/home/clark/.pm2/logs/ebook-out-0.log last 10 lines:
0|ebook    | info 2016-09-26 15:00:26 Forging enabled on account: 18314120909703123221
L
0|ebook    | info 2016-09-26 15:00:26 Forging enabled on account: 13969847998693785080
L
0|ebook    | info 2016-09-26 15:00:26 Forging enabled on account: 15336887350336188475
L
0|ebook    | info 2016-09-26 15:00:26 Forging enabled on account: 15106553597056061290
L
0|ebook    | info 2016-09-26 15:00:26 Forging enabled on account: 13719766223485644534
L
```

pm2重启应用

```
clark@E540:/data/blockchain/ebookcoin$ pm2 restart ebook
[PM2] Applying action restartProcessId on app [ebook](ids: 0)
[PM2] [ebook](0) ?
┌──────────┬────┬──────┬───────┬────────┬─────────┬────────┬─────┬───────────┬────────
──┐
│ App name │ id │ mode │ pid   │ status │ restart │ uptime │ cpu │ mem       │ watchin
g │
├──────────┼────┼──────┼───────┼────────┼─────────┼────────┼─────┼───────────┼────────
──┤
│ ebook    │ 0  │ fork │ 28059 │ online │ 1       │ 0s     │ 0%  │ 10.5 MB   │ disable
d │
└──────────┴────┴──────┴───────┴────────┴─────────┴────────┴─────┴───────────┴────────
──┘
 Use `pm2 show <id|name>` to get more details about an app
```

pm2重载应用

```
clark@E540:/data/blockchain/ebookcoin$ pm2 reload ebook
[PM2] Applying action reloadProcessId on app [ebook](ids: 0)
[PM2] [ebook](0) ?
┌──────────┬────┬──────┬───────┬────────┬─────────┬────────┬─────┬───────────┬────────
──┐
│ App name │ id │ mode │ pid   │ status │ restart │ uptime │ cpu │ mem       │ watchin
g │
├──────────┼────┼──────┼───────┼────────┼─────────┼────────┼─────┼───────────┼────────
──┤
│ ebook    │ 0  │ fork │ 28128 │ online │ 2       │ 0s     │ 0%  │ 10.5 MB   │ disable
d │
└──────────┴────┴──────┴───────┴────────┴─────────┴────────┴─────┴───────────┴────────
──┘
 Use `pm2 show <id|name>` to get more details about an app
```

pm2查看已启动的应用列表

```
clark@E540:/data/blockchain/ebookcoin$ pm2 list
┌──────────┬────┬──────┬───────┬────────┬─────────┬────────┬─────┬───────────┬────────
──┐
│ App name │ id │ mode │ pid   │ status │ restart │ uptime │ cpu │ mem       │ watchin
g │
├──────────┼────┼──────┼───────┼────────┼─────────┼────────┼─────┼───────────┼────────
──┤
│ ebook    │ 0  │ fork │ 28128 │ online │ 2       │ 41s    │ 0%  │ 86.6 MB   │ disable
d │
└──────────┴────┴──────┴───────┴────────┴─────────┴────────┴─────┴───────────┴────────
──┘
 Use `pm2 show <id|name>` to get more details about an app
```

pm2查看应用详情

```
clark@E540:/data/blockchain/ebookcoin$ pm2 show ebook
 Describing process with id 0 - name ebook
┌───────────────────┬──────────────────────────────────────────┐
│ status            │ online                                   │
│ name              │ ebook                                    │
│ restarts          │ 2                                        │
│ uptime            │ 59s                                      │
│ script path       │ /data/blockchain/ebookcoin/app.js        │
│ script args       │ N/A                                      │
│ error log path    │ /home/clark/.pm2/logs/ebook-error-0.log │
│ out log path      │ /home/clark/.pm2/logs/ebook-out-0.log   │
│ pid path          │ /home/clark/.pm2/pids/ebook-0.pid        │
│ interpreter       │ node                                     │
│ interpreter args  │ N/A                                      │
│ script id         │ 0                                        │
│ exec cwd          │ /data/blockchain/ebookcoin               │
│ exec mode         │ fork_mode                                │
│ node.js version   │ 0.12.15                                  │
│ watch & reload    │ ?                                        │
│ unstable restarts │ 0                                        │
│ created at        │ 2016-09-26T09:05:29.979Z                 │
└───────────────────┴──────────────────────────────────────────┘
 Revision control metadata
┌──────────────────┬────────────────────────────────────────────┐
│ revision control │ git                                        │
│ remote url       │ https://github.com/Ebookcoin/ebookcoin.git │
│ repository root  │ /data/blockchain/ebookcoin                 │
│ last update      │ 2016-09-26T09:06:26.562Z                   │
│ revision         │ f708fa4f523ec954b1a73694dec51ece7bb744c4   │
│ comment          │ Merge pull request #4 from robbinhan/master │
│ branch           │ master                                     │
└──────────────────┴────────────────────────────────────────────┘
 Code metrics value
┌────────────┬────────┐
│ Loop delay │ 0.12ms │
└────────────┴────────┘
 Add your own code metrics: http://bit.ly/code-metrics
 Use `pm2 logs ebook [--lines 1000]` to display logs
 Use `pm2 monit` to monitor CPU and Memory usage ebook
```

pm2停止应用

```
clark@E540:/data/blockchain/ebookcoin$ pm2 stop ebook
[PM2] Applying action stopProcessId on app [ebook](ids: 0)
[PM2] [ebook](0) ?
┌──────────┬────┬──────┬─────┬─────────┬─────────┬────────┬─────┬───────┬──────────┐
│ App name │ id │ mode │ pid │ status  │ restart │ uptime │ cpu │ mem   │ watching │
├──────────┼────┼──────┼─────┼─────────┼─────────┼────────┼─────┼───────┼──────────┤
│ ebook    │ 0  │ fork │ 0   │ stopped │ 2       │ 0      │ 0%  │ 0 B   │ disabled │
└──────────┴────┴──────┴─────┴─────────┴─────────┴────────┴─────┴───────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app
```

将应用从pm2中删除

```
clark@E540:/data/blockchain/ebookcoin$ pm2 delete ebook
[PM2] Applying action deleteProcessId on app [ebook](ids: 0)
[PM2] [ebook](0) ?
┌──────────┬────┬──────┬─────┬────────┬─────────┬────────┬─────┬─────┬──────────┐
│ App name │ id │ mode │ pid │ status │ restart │ uptime │ cpu │ mem │ watching │
└──────────┴────┴──────┴─────┴────────┴─────────┴────────┴─────┴─────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app
```

退出pm2

```
clark@E540:/data/blockchain/ebookcoin$ pm2 kill
[PM2] Stopping PM2...
[PM2][WARN] No process found
[PM2] All processes have been stopped and deleted
[PM2] PM2 stopped
```

Js对数据计算处理的各种问题

1.问题再现

(1)parseInt()最好不用

不要将 parseInt 当做转换 Number 和 Integer 的工具。

问题：默认，对小于 0.0000001 （1e-7） 的数字转换成 String 时，js会将其变成 科学记号法，再对这个数进行 parseInt 操作就会导致问题发生。即：

```
parseInt(0.0000008) === 8
```

解析：

常见的问题有浮点数比较：

```
console.log((0.1 + 0.2) == 0.3);  // false
console.log((0.1 + 0.2) === 0.3); // false
console.log(0.1 + 0.2); // 0.30000000000000004
```

再有：

```
parseInt(1000000000000000000000.5, 10); // 1
```

parseInt 的第一个类型是字符串，所以会将传入的参数转换成字符串，也就是 String(1000000000000000000000.5) 的结果为 '1e+21' 。 parseInt 并没有将 'e' 视为一个数字，所以在转换到就停止了。

这也就可以解释 parseInt(0.0000008) === 8 ：

```
String(0.000008);  // '0.000008'
String(0.0000008); // '8e-7'
```

参考： http://www.tuicool.com/articles/7nMbea

(2)对于大数据，**js**有限制

问题：

第三方api提供的数据：

```
{"content": "Hi", "created_at": 1340863646, "type": "text", "message_id": 575896550796
5321234, "from_user": "userC"}
```

其中message_id是19位number类型的。我用JSON.parse解析成JSON对象获取其中的信息，方法如下：

```
var jsonStr = '{"content": "Hi", "created_at": 1340863646, "type": "text", "message_id
": 5758965507965321234, "from_user": "userC"}';
var jsonObj = JSON.parse(jsonStr);
console.log(jsonObj.message_id);//得到结果是：5758965507965321000
```

得到的结果的最后三位变成000了。超过16位的数据解析后均会变为000；

解析：

浮点数范围：

```
as  large  as ±1.7976931348623157 × 10的308次方
as small as ±5 × 10的-324次方
```

精确整数范围：

```
The JavaScript number format allows you to exactly represent all integers between
-9007199254740992  and 9007199254740992 （即正负2的53次方）
```

数组索引还有位操作：

```
正负2的31次方
```

参考：

https://cnodejs.org/topic/4ff679c84764b7290214460a

https://cnodejs.org/topic/4fb3722c1975fe1e132b5a9a

2.终极答案

使用 node-bignum ，地址：https://github.com/Ebookcoin/node-bignum

社区出现了很多bignumber包，但只有node-bignum可以完美解决上述问题。诸如 bignumber.js 等无法解决小数问题（问题1）。

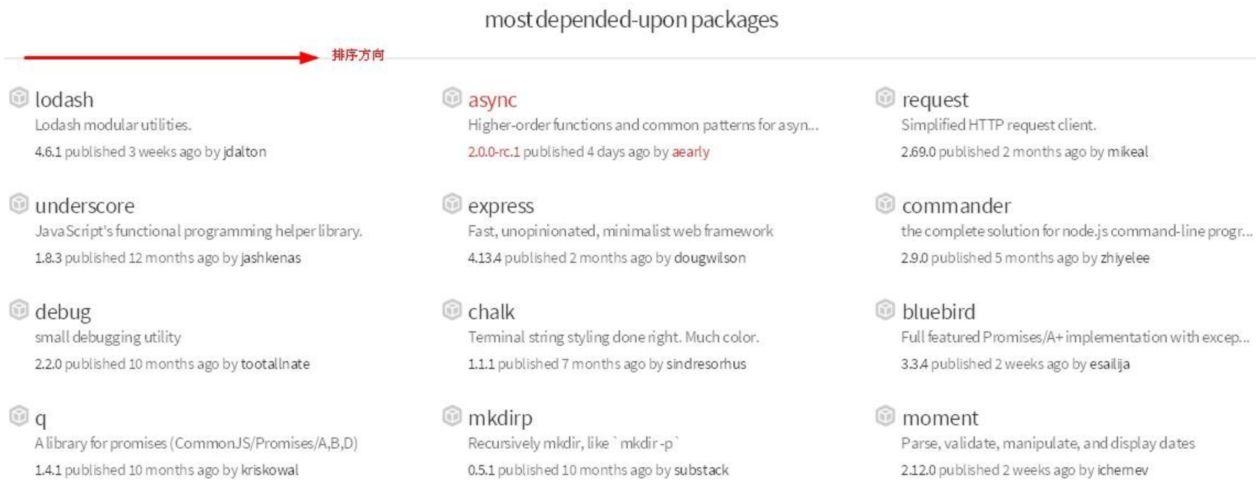
一张图学会使用Async组件进行异步流程控制

前言

前面说过，在Node.js的世界里“事事皆回调”，学习使用Node.js，最不可能回避的就是“回调”（用“调回”更直观些）。无法回避，自然要积极面对，因此开源社区出现了很多代码流程控制的解决方案。比如：bluebird，q，以及这里要图解的async。

这种基础性的技术，社区的文档极其丰富，但是我们为什么还要介绍？个人觉得，原因很简单，它真的很有必要，在只需要顺序编码的世界里，没有关于回调的操作流程或promise/a+规范（服务器帮助实现了），用不着大费周章。但是在Node.js的世界里，学习掌握一种方案，会显著提升编码能力。

为什么要介绍async，不是说bluebird性能更好吗？原因更简单：（1）Ebookcoin大量使用了aysnc，掌握它，对于理解和编码，事倍功半;（2）社区认可度高、文档丰富、使用简单、对代码没污染，无论是学习，还是使用，都没有风险。



这是async在 `https://npmjs.org` 上的依赖排名，除了lodash，就是它了。而且，bluebird和q也都在前10，也基本说明，使用流程控制组件是Node.js处理回调的标配。

概念定义

官方介绍：

`Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript.`

简单翻译：Async是一个为处理异步Js提供简单、直接、强大功能的实用模块。

流程类别

仅仅为了好区分、好记忆，简单汇总一下，全部流程应该是下面三种情况（并非官方描述）：

1.基本流程

从程序（多个函数）执行顺序的角度考量，包括顺序执行、并行执行、混合执行等基本流程。如果把函数间是否有依赖、是否限制函数执行的数量等，又可继续演化出很多种。

2.循环流程

基于某个条件的循环操作，根据条件使用方式，提供了诸多函数。

3.集合流程

上面都是针对一个或一组数据的处理。现实中，通常要对大批相同或相似的数据，比如：大批文件、地址，进行集中处理。显然，我们第一反应是循环调用上述流程即可，不过Async提供了对集合进行异步操作的工具方法（util），如forEach等，就叫它“集合流程”吧。

使用流程的概念可以明确告诉我们，当使用 `async.forEach()` 的时候，是在异步操作，与单纯的 `forEach` 方法调用是有区别的。

用法分类

async提供了70多个实用的函数，所有这些函数都遵守一个约定：你定义的异步函数（当作一个任务），最后一个参数必须是callback函数;该callback函数的第一个参数必须是Error，且callback函数调用一次。

这些函数大致分为3类，分别对应上面的3个类别：

1.基本型（一次多任务）

这对应基本流程和演化流程部分，对一项事务，多个任务的操作，调用形式为

```
async.funName(tasks, callback（error, results））
```

这里的funName，包括：series, parallel(parallelLimit), waterfall, auto(autoInject)等。这里的tasks可以是Array形式，也可以是Json形式，或者仅支持其中一种。

如果全部函数执行成功，callback里的results也会对应tasks的形式，将已经执行函数的结果转化为Array或Json形式;如果tasks中有一个函数出错，就会终止后续执行，调用callback，error就是该函数的错误信息，results或仅包含已经执行的结果、或同时包含未执行函数的结果占位符，或什么都不包含直接忽略。至于results具体是什么，一个 `log` 语句，自然就能知晓，不必去查文档。

2.循环型（多次单任务）

根据条件不同，可以使用下面的形式调用

```
async.funName(test, fn, callback)

或

async.doFunName(fn, test, callback)
```

这里的funName，包括：whilst（doWhilst），until（doUntil）， during（doDuring），retry（retryable），times（timesSeries, timesLimit）、forever（该函数的条件test就不用了，已经暗含条件）。

这里的条件值可以是表达式函数，或具体的次数。区分fn与test的顺序，很简单，只要用英文的意思去理解就可以，比如：async.doWhilst（），必然是先do，后whilst，因此参数自然是async.doWhilst(fn, test, callback)

3.集合型（多次单任务）

这个官方没有当作流程表述，当作集合操作方法提供的，是我个人的理解和杜撰。我觉得把它归为流程控制更好接受和理解。因此，仿造上面的循环型调用方法，只要将条件test改为一个集合就好，集合全部使用数组array形式。

```
async.funName(arr, iteratee, [callback])
```

这里的funName，包括：

```
each, eachSeries, eachLimit
forEachOf, forEachOfSeries, forEachOfLimit
map, mapSeries, mapLimit
filter, filterSeries, filterLimit
reject, rejectSeries, rejectLimit
reduce, reduceRight
detect, detectSeries, detectLimit
sortBy
some, someLimit, someSeries
every, everyLimit, everySeries
concat, concatSeries
```

当然，async还明确提供了其他几个流程控制，比如：compose，seq，applyEach（applyEachSeries），queue（priorityQueue），cargo，iterator，race等。

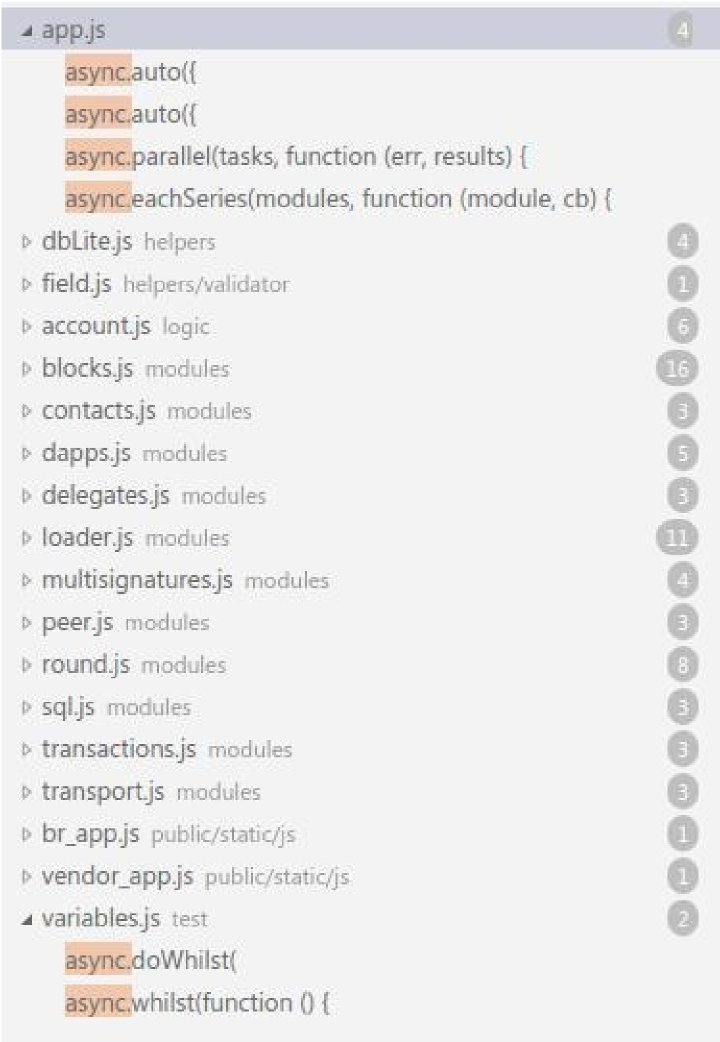
另外，还有几个Util（工具）函数，如：apply，nextTick，memoize，unmemoize，ensureAsync，constant，asyncify，wrapSync，log，dir，noConflict，timeout等，这些都能极大的方便代码撰写，改善代码性能。

脑图

上述解释和方法，我们全部放在一张脑图里，结合场景，按图索骥，能够很快找到正确的处理方法。

源码解读

看看 `Ebookcoin` 里面，async的使用吧。因为在《入口程序app.js解读》里，已经提及，这里就不说了，以后遇到再解释。



趣味实践

Async官方文档提供了很多实例， 简单直观。这里，我们不再举例，而是提出一个趣味问题，供思考：

问题：Aysnc能否用于递归调用，比如：某个爬虫程序，遍历某文件夹下全部文件信息的函数等？为什么？

总结

这又是一篇老生常谈的技术分享，但写完之后，我对异步操作的流程管理，更加清晰了。Aysnc很好，但也不是万能的，它对于那些反复自调用的代码就无能为力，因为限制任务的回调就是一次。后续，有机会还会继续深入学习研究它。

另外，参考里收集了几篇比较好的文章，建议读读。《Node.js最新技术栈之Promise篇》，作者 @i5ting 一位创业者，文章深入浅出，Promise/A+规范讲解循序渐进，是我喜欢的风格。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

参考

async官方文档

Node.js最新技术栈之Promise篇

JavaScript Promise迷你书（中文版）

关于时间戳及相关问题

前言

时间戳是很多应用系统，特别是加密货币开发设计中非常重要的元素。各种语言都提供了相应的时间处理函数，以前直接拿来就用了，也没有发现什么问题。但是在时间处理上，开发语言核心模块提供的个别Api并没有完全延续人类习惯。在Javascript语言里，有一个Date类的函数就非常奇葩，网络上很多文档的举例都是错误的，因此需要简单总结一下。

这似乎不是什么大问题，但是从stackoverflow.com网站上关于时间处理的问题（见参考）及受关注程度来看，却是很多人遇到的普遍问题。另外，万一还有更多“非常规”的用法存在，那么这种看似很小的问题，可能就是致命的（逻辑上没问题，意识中没发现）。按照前面关于异常和错误的处理要求，这种错误应该是程序员自身的问题了，唯一的解决办法就是提高能力，让自己更加严谨、更加认真。所以，这篇文章就是查缺补漏的，提醒以后别在这个小问题上翻船。

问题再现

请猜想下面的语句应该输出哪一天：

```
new Date(Date.UTC(2005,7,8));
```

结果是： `Mon Aug 08 2005 08:00:00 GMT+0800 (中国标准时间)` ，即： `2005年8月8日`

但是，看看这篇文档的解释 http://www.w3school.com.cn/jsref/jsref_utc.asp

其实，网上还有很多文档犯了这个错误。我本人也是在反复没有获得想要的结果情况下，才着手查询文档的。

时间戳的重要性

时间戳（timestamp），通常是一个字符序列，唯一地标识某一刻的时间，是指格林威治时间1970年01月01日00时00分00秒(北京时间1970年01月01日08时00分00秒)起至现在的总毫秒数。中国有国家授时中心，因其守时监测功能而保障时间戳证书中的时间的准确性和不被篡改，具有法律效力。

2008年11月25日，深圳市龙岗区法院依据最高法院“知识产权司法保护活动月”的要求公开宣判知识产权纠纷案，其中“利龙湖”一案系国内首例时间戳技术司法应用案例，宣判后双方当事人均未提起上诉，该案判决书已经发生法律效力。

从技术上讲，时间戳可以理解为数据写入或修改时的时间点，使用毫秒表示，是一项数据存在性证明（Proof of existence）的唯一有效参数。加密货币把时间戳作为重要字段信息，每笔交易、每个区块都要记录时间戳。对于亿书（请参考白皮书）而言，更是版权信息一个不可或缺的元素。

不同产品对时间处理的需求

我在思考这样一个问题：作为加密货币，亿书对每笔交易都有时间戳，而且对版权保护，时间戳还是非常重要的字段。如果，亿书一个节点在中国，另一个在美国，那么这个时间处理该如何统一，相关的方法该如何选择？

从理论上分析，不同产品可能对时间处理的需求也不相同。

1.对于一般的应用系统，特别是没有国际化要求的产品，时间自然没有特殊要求，前端后台统一是很简单的事，只要避免上面的失误，正确使用时间相关的接口函数即可。

2.对于有国际化需求的产品，特别是加密货币这种去中心化的产品，全世界的用户都在使用，就要考虑两个方面的因素，一个是面向用户的时间本地化问题，另一个是面向产品后台的时间统一性问题。

时间处理基本原理

天下人的时间都是一样的。但是，相同时刻，不同时区的人们，所处的时空是不一样的。不同时区，人们对于时空的感受又是那么惊奇的相似，比如：除了南北极，大多数地方都是早上五六点钟太阳升起，中午12点左右艳阳高照，下午五六点钟日落西山，如果一个中国人去了美国（比中国晚13个小时），原本是中午艳阳高照，一看手表，竟然是午夜0点钟，还不得精神分裂啊，因此为了迎合人类习惯，自然就有了很多的本地化需求。

从编程的角度看，时间具备唯一、均匀和恒定的特点，只要使用统一确定的参考点，就不至于在系统中出现混乱。所以，对应上面的产品需求，对于不需要国际化的产品，只要使用本地化时间处理方法即可（参考点是本地），前后端统一，逻辑简单。而对于国际化的产品，那就使用世界时间（参考点是国际日期变更线<见参考>，UTC或GMT标准时间），然后在客户端面向用户进行本地化处理即可。

Javascript语言的Date对象

js提供了操作日期和时间的对象Date。Date对象是js语言的一中内部数据类型（类函数），要使用语法new Date()创建。创建了对象之后，就可以使用多种方法来操作它了。这里列出几个需要熟练掌握的细节供参考。

（1）构造函数

```
new Date();
new Date(milliseconds);
new Date(dateString);
new Date(year, month[, day[, hour[, minutes[, seconds[, milliseconds]]]]]);
```

简单一句话：新建对象可以使用0～7个参数，这些参数可以是字符串或数字。

（2）参数

需要注意的是：

- Date时刻面向本地时间，参数代表的是本地时间，新建的对象代表的也是本地时间，输出的结果后面通常有`(某国标准时间)`等字样;
- 0参数，代表当前日期和时间;
- 1参数，若是数字milliseconds，应该是距离1970年1月1日午夜（UTC）的毫秒数，通常使用Date.UTC()函数来设定;若是字符串dateString，应该是时间格式，如：`new Date('December 17, 1995 03:24:00');`
- 2~7参数，只能是数字。只有`day`从*1*开始，其他都是从*0*开始。因为周期通常采取取模运算，所以如果数值大于合理范围（如月份为13），数值会被自动调整。比如： new Date(2013, 13, 1)等于new Date(2014, 1, 1)，它们都表示日期2014-02-01（注意月份是从0开始的）。
- 如果需要世界时，要使用 new Date(Date.UTC(...)) 和相同参数。

（3）方法

关于方法要注意两点：

- 静态方法：3个，都返回距离标准时间的毫秒数，分别对应构建函数三个形式，其中包括Date.now()、Date.parse()（解析一个表示日期的字符串）、Date.UTC()（从2到7参数）。

- 实例方法：4种，get类查询方法、set类设置方法、to类转换方法、of类，其中每种方法，都提供了本地和世界时两个参考系的方法，比如get[UTC]Date()方法。

特别注意的是：getTime()方法不分时区，返回Date对象的内部毫秒表示，这将是去伪存真的唯一途径，比如：如何判断不同时区是否同时呢，仅仅看这个方法的返回值就是了, 应该说在世界上不同时区，同时运行`(new Date()).getTime()`的返回值是一样的。

实践

编程里，需要处理的问题，无非是这样几个场景（下面的代码来自官方文档和亿书源码）：

（1）记录时间

```
var today = new Date(); //本地当前日期和时间
var birthday = new Date("1995-12-17T03:24:00"); //1995年12月17日
var birthday = new Date(1995,11,17); //1995年12月17日
```

如果需要格式化，就可以使用上面提到的to为前缀的转换方法，例如：toLocaleDateString()、toLocaleString()等，如果不满意，就找找现成的扩展插件，比如moment.js。如果仍不满意，那就自己实现吧，也很简单。

（2）计算时间

主要原则是计算距离标准时间的毫秒数，然后再计算。因为这类方法本身就是基于世界时UTC的，并且在计算过程中可能使用减法处理时间段（差值），因此参考点抵消，可以不去考虑参考点的存在。

这方面被大量用到项目中，也有很多第三方包存在，比如：timeago.js之类的，下面简单举例：

a> 在浏览器上推荐使用 Date 对象:

```
var start = Date.now();

// 这里进行耗时的方法调用:
doSomethingForALongTime();
var end = Date.now();
var elapsed = end - start; // 运行时间的毫秒值
```

为了兼容IE8及以前的浏览器，可以做如下处理：

```
if (!Date.now) {
    Date.now = function() { return new Date().getTime(); }
}
```

b> 在Node.js等环境下，建议使用内建的 `getTime()` 方法:

```
// 设置初始时间
var begin = Date.UTC(2016, 6, 1, 0, 0, 0, 0); // 2016年7月1日，月份从0开始

// 获得当前时间
var now = new Date();
var elapsed = now.getTime() - begin.getTime(); // 毫秒值
```

c> 为了获得秒为单位的时间戳，建议使用：

```
Math.floor(Date.now() / 1000)
```

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

电子书阅读： http://bitcoin-on-nodejs.ebookchain.org

参考

国际日期变更线

时间戳

JavaScript标准库 Date文档

How do you get a timestamp in JavaScript?

函数式编程入门经典

前言

虽然大家已经被面向对象编程（Object-oriented programing）洗脑了，但很明显这种编程方式在 JavaScript 里非常笨拙，这种语言里没有类可以用，社区采取的变通方法不下三种，还要应对忘记调用 `new` 关键字后的怪异行为，真正的私有成员只能通过闭包（closure）才能实现，而多数情况，就像我们在亿书代码里那样，把私有方法放在一个privated变量里，视觉上区分一下而已，本质上并非私有方法。对大多数人来说，函数式编程看起来才更加自然。而且，在Nodejs的世界里，大量的回调函数是数据驱动的，使用函数式编程更加容易理解和处理。

函数式编程远远没有面向对象编程普及，本篇文章借鉴了几篇优秀文档（见参考），结合亿书项目实践和个人体会，汇总了一些平时用得到的函数式编程思路，为更好的优化设计亿书做好准备。本篇内容包括函数式编程基本概念，主要特点和编码方法，其中的一些代码实例主要参考了《mostly adequate guide》，folktalejs 和 ramdajs 的相关代码，参考里也提供了它们的链接，请认真参考学习。如果想运行文中的代码，请提前安装ramda等相应的第三方组件。

什么是函数式编程？

简单说，“函数式编程”与“面向对象编程”一样，都是一种编写程序的方法论。它属于 “结构化编程” 的一种，主要思想是以数据为思考对象，以功能为基本单元，把程序尽量写成一系列嵌套的函数调用。

下面，我们就从一个简单的例子开始，来体会其中的奥妙和优势。这个例子来自于mostly-adequate-guide，作者说这是一个愚蠢的例子，并不是面向对象的良好实践，它只是强调当前这种变量赋值方式的一些弊端。这是一个海鸥程序，鸟群合并则变成了一个更大的鸟群，繁殖则增加了鸟群的数量，增加的数量就是它们繁殖出来的海鸥的数量。

（1）面向对象编码方式

```
var Flock = function(n) {
  this.seagulls = n;
};

Flock.prototype.conjoin = function(other) {
  this.seagulls += other.seagulls;
  return this;
};

Flock.prototype.breed = function(other) {
  this.seagulls = this.seagulls * other.seagulls;
  return this;
};

var flock_a = new Flock(4);
var flock_b = new Flock(2);
var flock_c = new Flock(0);

var result = flock_a.conjoin(flock_c).breed(flock_b).conjoin(flock_a.breed(flock_b)).s
eagulls;
//=> 32
```

按照正常的面向对象语言的编码风格，上面的代码好像没有什么错误，但运行结果却是错误的，正确答案是 `16` ，是因为 `flock_a` 的状态值 `seagulls` 在运算过程中不断被改变。别的先不说，如果 `flock_a` 的状态保持始终不变，结果就不会错误。

这类代码的内部可变状态非常难以追踪，出现这类看似正常，实则错误的代码，对整个程序是致命的。

（2）函数式编程方式

```
var conjoin = function(flock_x, flock_y) { return flock_x + flock_y };
var breed = function(flock_x, flock_y) { return flock_x * flock_y };

var flock_a = 4;
var flock_b = 2;
var flock_c = 0;

var result = conjoin(breed(flock_b, conjoin(flock_a, flock_c)), breed(flock_a, flock_b
));
//=>16
```

先不用考虑其他场景，至少就这个例子而言，这种写法简洁优雅多了。从数据角度考虑，逻辑简单直接，不过是简单的加（ `conjoin` ）和乘（ `breed` ）运算而已。

（3）函数式编程的延伸

函数名越直白越好，改一下：

```
var add = function(x, y) { return x + y };
var multiply = function(x, y) { return x * y };

var flock_a = 4;
var flock_b = 2;
var flock_c = 0;

var result = add(multiply(flock_b, add(flock_a, flock_c)), multiply(flock_a, flock_b))
;
//=>16
```

这么一来，你会发现我们还能运用小学都学过的运算定律：

```
// 结合律
add(add(x, y), z) == add(x, add(y, z));

// 交换律
add(x, y) == add(y, x);

// 同一律
add(x, 0) == x;

// 分配律
multiply(x, add(y,z)) == add(multiply(x, y), multiply(x, z));
```

我们来看看运用这些定律如何简化这个海鸥小程序：

```
// 原有代码
add(multiply(flock_b, add(flock_a, flock_c)), multiply(flock_a, flock_b));

// 应用同一律，去掉多余的加法操作（add(flock_a, flock_c) == flock_a）
add(multiply(flock_b, flock_a), multiply(flock_a, flock_b));

// 再应用分配律
multiply(flock_b, add(flock_a, flock_a));
```

到这里，程序就变得非常有意思，如果更加复杂的应用，也能够确保结果可以预期，这就是函数式编程。

函数式编程的优势

1.易于开发：代码简洁，大大降低开发成本

从上面的代码，可以体会到这一点。使用函数式编程，可以充分发挥Javascript语言自身的优点，每一个函数都是独立单元，便于调试和测试，也方便模块化组合，代码量少、开发效率高。有人比较过C语言与Lisp语言，同样功能的程序，极端情况下，Lisp代码的长度可能是C代码的二十分之一。

2.易于分享：更接近自然语言，代码即文档

上面使用分配律之后的代码 `multiply(flock_b, add(flock_a, flock_a))` ，完全可以改成下面这样：

```
add(flock_a, flock_a).multiply(flock_b) // = (flock_a + flock_a) * flock_b
```

特别是下面这样的句式，如果是用惯了ruby on rails的小伙伴，更加熟悉下面的语句：

```
User.all().sortBy('name').limit(20)
```

3.性能更高：能够实现"并发编程"（concurrency）

函数式编程不依赖、也不会改变外界的状态，相同输入获得相同输出，因此不存在"锁"线程的问题。不必担心一个线程的数据，被另一个线程修改，所以可以很放心地把工作分摊到多个线程，实现"并发编程"。目前的计算机基本上都是多核的了，多线程应用将是常态，亿书客户端也会考虑优化线程服务，提高软件性能，改善用户体验，这将非常有帮助。

4.部署简单：热部署和热升级

函数式编程没有副作用，只要接口没变化，改变函数内部代码对外部没有任何影响。所以，可以在运行状态下直接升级代码，不需要重启，也不需要停机。这对于每秒要处理很多交易的加密货币系统来说，尤为重要。亿书将在未来实现每个节点的热部署和热升级，使节点建立和维护零难度。

函数式编程的基本原则

总的来说，就是“正确地写出正确的函数”。这话有点绕，不过我们写了太多面向对象的代码，可能已被“毒害”太深，不得不下点猛药，说点狠话，不然记不住。函数式编程，当然函数是主角，被称为“一等公民”，特别是对于 JavaScript 语言来说，可以像对待任何其他数据类型一样对待——把它们存在数组里，当作参数传递，赋值给变量...等等。但是，说起来容易，真正做起来，并非每个人都能轻松做到。下面是写出正确函数的几个原则：

（1）直接把函数赋值给变量

记住：凡是使用 `return` 返回函数调用的，都可以去掉这个间接包裹层，最终连参数和括号一起去掉！

以下代码都来自 npm 上的模块包：

```
// 太傻了
var getServerStuff = function(callback){
  return ajaxCall(function(json){
    return callback(json);
  });
};

// 这才像样
var getServerStuff = ajaxCall;
```

世界上到处都充斥着这样的垃圾 ajax 代码。以下是上述两种写法等价的原因：

```
// 这行
return ajaxCall(function(json){
  return callback(json);
});

// 等价于这行
return ajaxCall(callback);

// 那么，重构下 getServerStuff
var getServerStuff = function(callback){
  return ajaxCall(callback);
};

// ...就等于
var getServerStuff = ajaxCall; // <-- 看，没有括号哦
```

（2）使用最普适的方式命名

函数属于操作，命名最好简单直白体现功能性，比如 `add` 等。参数是数据，最好不要限定在特定的数据上，比如 `articles` ，就能让写出来的函数更加通用，避免重复造轮子。例如：

```
// 只针对当前的博客
var validArticles = function(articles) {
  return articles.filter(function(article){
    return article !== null && article !== undefined;
  });
};

// 对未来的项目友好太多
var compact = function(xs) {
  return xs.filter(function(x) {
    return x !== null && x !== undefined;
  });
};
```

（3）避免依赖外部变量

不依赖外部变量和环境，就能确保写出的函数是 `纯函数` ，即：相同输入得到相同输出的函数。 比如 `slice` 和 `splice` ，这两个函数的作用一样，但方式不同， `slice` 符合纯函数的定义是因为对相同的输入它保证能返回相同的输出。而 `splice` 却会嚼烂调用它的那个数组，然后再吐出来，即这个数组永久地改变了。

再如：

```
// 不纯的
var minimum = 21;

var checkAge = function(age) {
  return age >= minimum;
};


// 纯的
var checkAge = function(age) {
  var minimum = 21;
  return age >= minimum;
};
```

在上面不纯的版本中， `checkAge` 的结果取决于 `minimum` 这个外部可变变量的值（系统状态值）。输入值之外的因素能够左右 `checkAge` 的返回值，产生很多副作用，不仅让它变得不纯，而且导致每次我们思考整个软件的时候都将痛苦不堪，本法就会产生很多bug。这些副作用包括但不限于：更改文件系统、往数据库插入记录、发送一个 http 请求、打印log、获取用户输入、DOM 查询、访问系统状态等，总之，就是在计算结果的过程中，导致系统状态发生变化，或者与外部世界进行了可观察的交互。

在数学领域，函数是这么定义的（请参考 百度百科上函数定义）：一般的，在一个变化过程中，假设有两个变量x、y，如果对于任意一个x都有唯一确定的一个y和它对应，那么就称y是x的函数。直白一点就是， `函数是不同数值之间的特殊关系：对每一个输入值x返回且只返回一个输出值y` 。从这个定义来看，我所谓的 `纯函数` 其实就是数学函数。这会给我们带来可缓存性、可移植性、自文档化、可测试性、合理性、并行代码等很多好处（具体实例请参考[mostly-adequate-guide（英文）][]）。

（4）面对 `this` 值，小心加小心

`this` 就像一块脏尿布，尽可能地避免使用它，因为在函数式编程中根本用不到它。然而，在使用其他的类库时，你却不得不低头。如果一个底层函数使用了 `this` ，而且是以函数的方式被调用的，那就要非常小心了。比如：

```
var fs = require('fs');

// 太可怕了
fs.readFile('freaky_friday.txt', Db.save);

// 好一点点
fs.readFile('freaky_friday.txt', Db.save.bind(Db));
```

把 Db 绑定（bind）到它自己身上以后，你就可以随心所欲地调用它的原型链式垃圾代码了。

怎样进行函数式编程？

这里汇总一些常用的工具和方法。

（1）柯里化（curry）：动态产生新函数

什么是柯里化？维基百科（见参考）的解释是，柯里化是指把接受多个参数的函数变换成接受一个单一参数（第一个）的函数，返回接受剩余参数的新函数的技术。通俗点说，只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数。

这里有两个关键，首先，明确规则。前面的参数（可以是函数）是规则，相当于新函数的私有变量（通过类似闭包的方式）；其次，明确目的。目的是获得新函数，而这个新函数才是真正用来处理业务数据的，所以与业务数据相关的参数，最好放在后面。

这是函数式编程的重要技巧之一，在使用各类高阶函数（参数或返回值为函数的函数）的时候非常常见。通常大家不定义直接操作数组的函数，因为只需内联调用 `map`、`sort`、`filter` 以及其他的函数就能达到目的，这时候多会用到这种方法 。

最简单的例子：

```
var add = function(x) {
  return function(y) {
    return x + y;
  };
};

var addTenTo = add(10);

addTenTo(2);
// 12
```

这里是自定义的 `add` 函数，它接受一个参数并返回一个新的函数。调用 `add` 之后，返回的函数就通过闭包的方式记住 `add` 的第一个参数。我们还可以借助工具使这类函数的定义和调用更加容易。比如，利用 `lodash` 包的 `curry` 方法，上面的 `add` 方法就变成这样：

```
var curry = require('lodash').curry;

var add = curry(function(x, y) {
    return x + y;
});
```

下面看几个更实用的例子：

```
var curry = require('lodash').curry;

var match = curry(function(what, str) {
  return str.match(what);
});

// 1.当普通函数调用
match(/\s+/g, "hello world");
// [ ' ' ]

match(/\s+/g)("hello world");
// [ ' ' ]

// 2.使用柯里化的新函数
var hasSpaces = match(/\s+/g);
// function(x) { return x.match(/\s+/g) }

hasSpaces("hello world");
// [ ' ' ]

hasSpaces("spaceless");
// null

// 3.大胆嵌套使用
var filter = curry(function(f, ary) {
  return ary.filter(f);
});

var findHasSpacesOf = filter(hasSpaces);
// function(xs) { return xs.filter(function(x) { return x.match(/\s+/g) }) }

findHasSpacesOf(["tori_spelling", "tori amos"]);
// ["tori amos"]
```

（2）组合（compose）：自由组合新函数

`组合` 就是把两个函数结合起来，产生一个崭新的函数。这符合范畴学的组合理论，也就是说组合某种类型（本例中是函数）的两个元素还是会生成一个该类型的新元素，就像把两个乐高积木组合起来绝不可能得到一个林肯积木一样。`组合` 函数的代码非常简单，如下：

```
var compose = function(f,g) {
  return function(x) {
    return f(g(x));
  };
};
```

`f` 和 `g` 都是函数，`x` 是在它们之间通过"管道"传输的值。`g` 先于 `f` 执行，因此数据流是从右到左的，这符合数学上的"结合律"的概念，能为编码带来极大的灵活性（可以任意拆分和组合），而且不用担心执行结果出现意外。比如：

```
// 结合律（associativity）
var associative = compose(f, compose(g, h)) == compose(compose(f, g), h);
// true
```

函数式编程有个叫"隐式编程"（[Tacit programming][]，见参考）的概念，也叫"point-free 模式"，意思是函数不用指明操作的参数（也叫 points），而是让组合它的函数处理参数。柯里化以及组合协作起来非常有助于实现这种模式。首先，利用柯里化，让每个函数都先接收数据，然后操作数据；接着，通过组合，实现把数据从第一个函数传递到下一个函数那里去。这样，就能做到通过管道把数据在接受单个参数的函数间传递。

显然，隐式编程模式隐去了不必要的参数命名，让代码更加简洁和通用。通过这种模式，我们也很容易了解一个函数是否是接受输入返回输出的小函数。比如，replace，split等都是这样的小函数，可以直接组合；map接受两个参数自然不能直接组合，不过可以先让它接受一个函数，转化为一个参数的函数；但是 while 循环是无论如何不能组合的。另外，并非所有的函数式代码都是这种模式的，所以，适当选择，不能使用的时候就用普通函数。

下面，看个例子：

```
// 非隐式编程，因为提到了数据：word
var snakeCase = function (word) {
  return word.toLowerCase().replace(/\s+/ig, '_');
};

// 隐式编程
var snakeCase = compose(replace(/\s+/ig, '_'), toLowerCase);
```

本来，js的错误定位就不准确，这样的组合给 debug 带来了更多困难。还好，我们可以使用下面这个实用的，但是不纯的 `trace` 函数来追踪代码的执行情况。

```
var trace = curry(function(tag, x){
  console.log(tag, x);
  return x;
});

var dasherize = compose(join('-'), toLower, split(' '), replace(/\s{2,}/ig, ' '));

dasherize('The world is a vampire');
// TypeError: Cannot read property 'apply' of undefined
```

这里报错了，来 `trace` 下：

```
var dasherize = compose(join('-'), toLower, trace("after split"), split(' '), replace(
/\s{2,}/ig, ' '));
// after split [ 'The', 'world', 'is', 'a', 'vampire' ]
```

啊，`toLower` 的参数是一个数组（记住，上面的代码是从右向左执行的奥），所以需要先用 `map` 调用一下它。

```
var dasherize = compose(join('-'), map(toLower), split(' '), replace(/\s{2,}/ig, ' '))
;

dasherize('The world is a vampire');

// 'the-world-is-a-vampire'
```

（3）注释：签名函数的行为和目的

函数式编程非常灵活，包括隐式编程在内，参数被大大减少和弱化，这就为我们阅读和使用函数带来困扰，对协作开发也是一项挑战，如何解决？通常的做法就是添加详细的文档或注释，不过函数式编程有其自己的处理方式，那就是类型签名。类型签名在写纯函数时所起的作用非常大，短短一行，就能暴露函数的行为和目的。这里介绍一下，在函数式编程里，非常著名的Hindley-Milner类型系统。这个名称是两位科学家的名字组合，常被简称为HM类型系统。在使用该类型系统中，注意以下几点要素：

- 函数都写成 **`a -> b`** 的样子。其中 `a` 和 `b` 是任意类型的变量，这句话的意思是“一个接受a，返回b的函数”。延伸一下，`a -> (b -> c)` 的意思就是“一个接受a，返回一个接受b返回c的函数的函数”，即柯里化了。
- 把最后一个类型视作返回值。比如 `a -> (b -> c)` 简单的理解成 `a -> b -> c` 也没有关系，只不过中间柯里化的过程被忽略了而已。
- 参数优先级是从左向右，每传一个参数，就会得到后面对应部分的类型签名。比如，`a -> (b -> c)`，传入了参数 a，得到的自然是新函数，`b -> c` 。

- 可以在类型签名中使用变量。把变量命名为 `a` 和 `b` 只是一种约定俗成的习惯，可以使用任何自己喜欢的名称。对于相同的变量名，其类型也一定相同。比如：`a -> b` 可以是从任意类型的 `a` 到任意类型的 `b` ，但是 `a -> a` 必须是同一个类型。例如，可以是 `String -> String` ，也可以是 `Number -> Number` ，但不能是 `String -> Bool` 。同时，也说明函数将会以一种统一的行为作用于所有的类型 *a* 或 *b* ，而不能做任何特定的事情，这能够帮助我们推断函数可能的实现。

最简单的例子：

```
//  capitalize :: String -> String
var capitalize = function(s){
  return toUpperCase(head(s)) + toLowerCase(tail(s));
}

capitalize("smurf");
//=> "Smurf"
```

这里， `capitalize` 函数的类型签名就是“capitalize :: String -> String”这行注释，可以理解为“一个接受 `String` 返回 `String` 的函数”。通俗点说，它接受一个 `String` 类型作为输入，并返回一个 `String` 类型的输出。

复杂一点的例子：

```
//  match :: Regex -> String -> [String]
//  理解为：接受一个 `Regex` 和一个 `String`，返回一个 `[String]`
var match = curry(function(reg, s){
  return s.match(reg);
});

//  match :: Regex -> (String -> [String])
//  理解为：接受一个 `Regex` 作为参数，返回一个从 `String` 到 `[String]` 的函数
var match = curry(function(reg, s){
  return s.match(reg);
});

//  onHoliday :: String -> [String]
//  理解为：已经调用了 `Regex` 参数的 `match`。给了第一个参数 `Regex`，返回结果就是后面的签名内容
var onHoliday = match(/holiday/ig);
```

让我们体会类型签名的好处：

```
// head :: [a] -> a
compose(f, head) == compose(head, map(f));

// filter :: (a -> Bool) -> [a] -> [a]
compose(map(f), filter(compose(p, f))) == compose(filter(p), map(f));
```

上面第一个等式，左边：先获取数组的`头部`，然后对它调用函数`f`；右边：先对数组中的每一个元素调用`f`，然后再取其返回结果的`头部`。尽管没有看到head，f等具体的代码实现，我们也知道这两个表达式的作用显然是相等的，但是前者要快得多，这是常识。这就为我们使用和优化提供了便利，在使用函数的时候，可以更加灵活的选择。

（4）容器：处理控制流、异常、异步和状态的独立模块

从代码功能来说，类型是最小单元，类似原子；函数就是基本单元，类似由原子形成的各类细胞；容器就是由细胞构成的人体组织。在一个程序里，容器就像一个功能模块（比面向对象中的类要灵活得多），有自己的上下文，包含了特定的方法（函数，主要是仿函数）和属性（状态），能够独立完成某方面的功能或事务。容器之间的操作和通信，需要用到特殊的数据类型——仿函数（functor）。

首先，创建一个容器：

```
var Container = function(x) {
  this.__value = x;
}

Container.of = function(x) { return new Container(x); };

// (a -> b) -> Container a -> Container b
Container.prototype.map = function(f){
  return Container.of(f(this.__value))
}
```

我们把它命名为`Container`，使用`Container.of`作为构造器（constructor），这样就不用到处去写糟糕的`new`关键字了，非常省心。这个容器具备一切容器的标准特征：

- 容器有且只有一个属性的对象。尽管容器可以有不止一个的属性，但大多数容器还是只有一个。我们很随意地把`Container`的这个属性命名为`__value`，你也可以改成别的。所以说，容器就像面向对象的类，但不是，因为我们不会为它添加面向对象观念下的属性和方法。
- 容器必须能够装载任意类型的值。因此，这里的`__value`不能是某个特定的类型。
- **of**方法是容器的入口访问方法。这是一种简单通用地往仿函数里填值的方式。数据一旦存放到容器，就会一直待在那儿。我们可以用`.__value`获取到数据，但尽量别这么做。
- 仿函数是容器的出口访问方法。操作和使用数据要使用仿函数，因此，在具体使用中，仿函数基本上代表了容器本身。在函数式编程里，到处都有仿函数的身影，像 tree、list、map 和 pair 等可迭代数据类型，以及eventstream 和 observable 也都是。

什么是仿函数？

网上搜索了一下，大家普遍认为这个概念很难理解。说实话，从字面意思上，无论是中文函子、仿函数，还是英文functor，我一开始都没有理解是个什么东西。 大家普遍认可的是 Functor, Applicative, 以及 Monad 的图片阐释 里的解释，图文并茂非常清晰，请自行查阅吧。

对这种舶来品，我通常的做法是直接看它的英文解释（见参考 维基百科上的functor词条），`In mathematics, a functor is a type of mapping between categories, which is applied in category theory.` 译为：在数学中，仿函数应用于范畴学，是一种范畴之间的映射。按照这里的解释把它称为 `Mappable` 或许更为恰当。

接着上面的容器话题，一旦容器里有了值，不管这个值是什么，我们就需要一种方法来让别的函数能够操作它。这句话的意思是，值外面有了容器，就给定了一个范畴（上下文），我们就没有办法像前面那样简单的操作这个值了，怎么办？仿函数就派上用场了。形象点说，也就是容器设定了一个范畴，仿函数就为它搭建了一个交流通道。

上面代码里的 `map` 跟数组那个著名的 `map` 一样，除了前者的参数是 `Container a` 而后者是 `[a]` 。它们的使用方式也几乎一致：

```
// 非特殊情况，我们使用 Ramdajs
var _ = require('ramda');

Container.of(2).map(function(two){ return two + 2 })
//=> Container(4)

Container.of("flamethrowers").map(function(s){ return s.toUpperCase() })
//=> Container("FLAMETHROWERS")

Container.of("bombs").map(concat(' away')).map(_.prop('length'))
//=> Container(10)
```

`of` 方法不单单是用来避免使用 `new` 关键字的，而是用来把值放到默认最小化上下文（default minimal context）中的。 `of` 其实就是 `pure` 、 `point` 、 `unit` 和 `return` 之类的函数，它是一个称之为 *pointed*仿函数 （实现了 `of` 方法的仿函数）的重要接口的一部分。这里的关键是把任意值丢到容器里然后开始到处使用 `map` 的能力。

我们能够在不离开 `Container` 的情况下操作容器里面的值。 `Container` 里的值传递给 `map` 函数之后，就可以任我们操作；操作结束后，为了防止意外再把它放回它所属的 `Container` 。这样做的结果是，我们能连续地调用 `map` ，运行任何我们想运行的函数，甚至还可以改变值的类型。

让容器自己去运用函数有利于对函数运用的抽象。当 `map` 一个函数的时候，我们请求容器来运行这个函数，这是一种十分强大的理念。这种让容器去运行函数的方法就是“仿函数”。通俗点讲，一个函数在调用的时候，如果被 `map` 包裹了，那么它就会从一个非仿函数转换为一个

仿函数。一般情况下，普通函数更适合操作普通的数据类型而不是容器类型，在必要的时候再通过 map 变为合适的仿函数去操作容器类型，这样做的好处是随需求而变，能得到更简单、重用性更高的函数。

仿函数的概念既然来自于范畴学，应该满足一些定律，学习理解这些实用的定律，会帮助我们更好的使用它。两个重要的定律：

```
// 同一性 identity
map(id) === id;

// 结合律 composition
compose(map(f), map(g)) === map(compose(f, g));
```

下面，根据功能不同，介绍几种重要的仿函数。

1> 数据验证（**Maybe**）

上面的 `Container` 功能单一，下面，让我们对它进行一系列重构，以满足我们不同的需要。请注意，每次重构，得出的新容器，都是一种具备特定功能的仿函数。先给数据加上验证看看，毕竟数据是否为空，是编程无法绕开的逻辑之一，具备这种特性的仿函数，函数式编程里称之为Maybe。更多内容，请参考学习 folktale's data.maybe。

```
var Maybe = function(x) {
  this.__value = x;
}

Maybe.of = function(x) {
  return new Maybe(x);
}

Maybe.prototype.isNothing = function() {
  return (this.__value === null || this.__value === undefined);
}

Maybe.prototype.map = function(f) {
  return this.isNothing() ? Maybe.of(null) : Maybe.of(f(this.__value));
}

Maybe.of(null).map(match(/a/ig));
//=> Maybe(null)

Maybe.of({name: "Boris"}).map(_.prop("age")).map(add(10));
//=> Maybe(null)
```

看似很小的改进，却让代码更加健壮。除了时刻检查参数的存在性，还能用 `Maybe(null)` 来发送失败的信号，让程序接到这个信号时立刻切断后续代码的执行，这通常是编码希望达到的效果。

2> 错误处理（Either）

对于错误处理，当我们不知道可能是什么错误时，用的比较多的是 `throw/catch` ，这会影响性能，错误信息也不怎么友好，而且可能会打破“纯”函数，让其变得不再“纯”。当然，我们也会使用if语句来判定，给出具体的错误信息。在函数式编程里，通常使用 `Either` 这个仿函数。字面含义为“或者”的意思，属于二选一的两个分支。更多内容，请参考学习 folktale's data.either。

这里，列出部分代码，如下：

```
var Left = function(x) {
  this.__value = x;
}

Left.of = function(x) {
  return new Left(x);
}

Left.prototype.map = function(f) {
  return this;
}

var Right = function(x) {
  this.__value = x;
}

Right.of = function(x) {
  return new Right(x);
}

Right.prototype.map = function(f) {
  return Right.of(f(this.__value));
}
```

这里略去了创建 `Either` 父类，只给出了它的两个子类 `Left` （代表错误） 和 `Right` （代表正确），来看看它们是怎么运行的：

```
Right.of("rain").map(function(str){ return "b"+str; });
// Right("brain")

Left.of("rain").map(function(str){ return "b"+str; });
// Left("rain")

Right.of({host: 'localhost', port: 80}).map(_.prop('host'));
// Right('localhost')

Left.of("rolls eyes...").map(_.prop("host"));
// Left('rolls eyes...')
```

`Left` 无视 `map` 它的请求。`Right` 的作用就是一个最基础的 `Container` 。这里强大的地方在于，`Left` 有能力在它内部嵌入一个错误消息。

前面说了，我们可以用 `Maybe(null)` 来表示失败并把程序引向另一个分支，但是它不会告诉我们太多信息，有时候我们想知道失败的原因是什么。比如：

```
var moment = require('moment');

//  getAge :: Date -> User -> Either(String, Number)
var getAge = _.curry(function(now, user) {
  //moment v2.3.0 以后的版本要添加true参数，表示使用严格模式，参考：http://momentjs.com/docs/#
/parsing
  var birthdate = moment(user.birthdate, 'YYYY-MM-DD', true);
  if(!birthdate.isValid()) return Left.of("Birth date could not be parsed");
  return Right.of(now.diff(birthdate, 'years'));
});

getAge(moment(), {birthdate: '2005-12-12'});
// Right(9)

getAge(moment(), {birthdate: '20010704'});
// Left("Birth date could not be parsed")
```

这么一来，就像 `Maybe(null)` ，当返回一个 `Left` 的时候就直接让程序终止。跟 `Maybe(null)` 不同的是，现在我们对程序为何终止有了更多信息。让我们进一步看看，怎么用：

```
//  fortune :: Number -> String
var fortune  = _.compose(_.concat("If you survive, you will be "), _.add(1));

//  zoltar :: User -> Either(String, _)
// 在类型签名中使用 `_`， 表示一个应该忽略的值；
// 通常，不会把 `console.log` 放到 `zoltar` 函数里，而是在调用 `zoltar` 的时候才 `map` 它
var zoltar = _.compose(_.map(console.log), _.map(fortune), getAge(moment()));

zoltar({birthdate: '2005-12-12'});
// "If you survive, you will be 11"
// Right(undefined)

zoltar({birthdate: 'balloons!'});
// Left("Birth date could not be parsed")
```

进一步优化，添加一个 either 方法，让它接受两个函数（分别对应left 和 right的情况）和一个静态值为参数：

```
//  either :: (a -> c) -> (b -> c) -> Either a b -> c
var either = _.curry(function(f, g, e) {
  switch(e.constructor) {
    case Left: return f(e.__value);
    case Right: return g(e.__value);
  }
});

//  zoltar :: User -> _
var zoltar = _.compose(console.log, either(_.identity, fortune), getAge(moment()));

zoltar({birthdate: '2005-12-12'});
// "If you survive, you will be 10"
// undefined

zoltar({birthdate: 'balloons!'});
// "Birth date could not be parsed"
// undefined
```

`Either` 并不仅仅只对合法性检查这种一般性的错误有作用，对一些更严重的、能够中断程序执行的错误，比如文件丢失或者 socket 连接断开等，同样效果显著。另外，这里仅把 `Either` 当作一个错误消息的容器，其实它还能做更多的事情。

3> 异步处理（**Task**）

在JavaScript的世界里，回调（callback）是无法回避的。还好，处理异步代码，函数式编程有一种更好的方式。这种方式的内部机制比较复杂，所以还是通过使用 Folktale 里的 `Data.Task` ，来从实例中去体会吧：

```
// 这是Folktale官方的例子
var Task = require('data.task')
var fs = require('fs')

// read : String -> Task(Error, Buffer)
function read(path) {
  return new Task(function(reject, resolve) {
    fs.readFile(path, function(error, data) {
      if (error)  reject(error)
      else        resolve(data)
    })
  })
}

// decode : Task(Error, Buffer) -> Task(Error, String)
function decode(buffer) {
  return buffer.map(function(a) {
    return a.toString('utf-8')
  })
}

var one = decode(read('one.txt'))
var two = decode(read('two.txt'))

// 使用 `.chain` 来关联两个异步行为，使用`.map` 同步运算任务的最终值
var concatenated = one.chain(function(a) {
                     return two.map(function(b) {
                       return a + b
                     })
                   })

// 必须使用 `.fork` 来明确执行，错误使用第一个函数抛出，成功就调用第二个函数
concatenated.fork(
  function(error) { throw error },
  function(data)  { console.log(data) }
)
```

例子中的 `reject` 和 `resolve` 函数分别是失败和成功的回调。我们用 `chain` 来运行两个异步行为；简单地调用 `Task` 的 `map` 函数，就能操作异步执行之后的值，好像这个值就在那儿似的；调用 `fork` 方法才能运行 `Task`，它会 fork 一个子进程运行它接收到的参数代码，其他部分的执行不受影响，主线程也不会阻塞。

这里的控制流是线性的。我们只需要从下读到上，从右读到左就能理解代码，即便这段程序实际上会在执行过程中到处跳来跳去。这种方式使得阅读和理解应用程序的代码比那种要在各种回调和错误处理代码块之间跳跃的方式容易得多。

4> 嵌套处理（Monad 和 Applicative）

这个世界是复杂的，我们把普通函数放进了容器，弄出这么多仿函数出来。那如果把容器放进相同的容器里，一层层嵌套起来（就像层层包裹的洋葱Monad）怎么办呢？最直接的处理方式就是多用几次map就可以了。不过，这类情况，有专门的处理方式：

- 同类型容器嵌套调用 —— Monad

如果有两层相同类型的嵌套，那么就可以用 `join` 把它们压扁到一块去。这种结合的能力，仿函数之间的关联，就是 monad 之所以成为 monad 的原因。来看看它更精确的完整定义：

> monad 是可以变扁（flatten）的 pointed 仿函数。

一个仿函数，只要它定义了一个 `join` 方法和一个 `of` 方法，并遵守一些定律，那么它就是一个同类型嵌套仿函数（这是我个人杜撰的名称，眞正函数式编程的高手会不喜欢，大家习惯的还是monad）。`join` 的实现并不复杂，来为 `Maybe` 定义一个：

```
Maybe.prototype.join = function() {
  return this.isNothing() ? Maybe.of(null) : this.__value;
}
```

如果有一个 `Maybe(Maybe(x))` ，那么 `.__value` 将会移除多余的一层，然后就能安心地从那开始进行 `map` 。要不然，将会只有一个 `Maybe` ，因为从一开始就没有任何东西被 `map` 调用。

很多情况下，总是在紧跟着 `map` 的后面调用 `join` 。这时候，可以把这个行为进一步抽象到一个叫做 `chain` 的函数里：

```
//  chain :: Monad m => (a -> m b) -> m a -> m b
var chain = curry(function(f, m){
  return m.map(f).join(); // 或者 compose(join, map(f))(m)
});
```

这里仅仅是把 map/join 打包到一个单独的函数中。在一些库里， `chain` 也叫做 `>>=` （读作 bind）或者 `flatMap` ，都是同一个概念的不同名称罢了。 `flatMap` 是最简单明了，但 `chain` 是 JS 里接受程度最高的一个。虽然这么简单的处理了一下，该方法的用途却被无限扩展，我们上面使用的“Task”的例子用的就是它，请访问它的源码自行研究吧。

- 不同类型容器的调用 —— Applicative

同类型容器嵌套，使用Monad仿函数处理方式，这种方式有一个问题，那就是顺序执行问题：所有的代码都只会在前一个函数执行完毕之后才执行。其实，在很多情况下，这是没有必要的。其更好的替代方法就是使用applicative仿函数。applicative仿函数是实现了 `ap` 方法的pointed仿函数。 `ap` 方法，类似下面这样：

```
Container.prototype.ap = function(other_container) {
  return other_container.map(this.__value);
}
```

`this.__value` 是一个函数，将会接收另一个 functor 作为参数，所以我们只需 `map` 它。这里隐含的定律是：

```
F.of(x).map(f) == F.of(f).ap(F.of(x))
```

map 一个 `f` 等价于 `ap` 一个值为 `f` 的 functor。更简单的理解就是: `map` 等价于 `of/ap` 。那么，可以利用这点来定义 `map` ：

```
// 从 of/ap 衍生出的 map
X.prototype.map = function(f) {
  return this.constructor.of(f).ap(this);
}
```

如果已经有了一个 `chain` 函数，我们也可以借助 monad 轻松得到仿函数（含有map方法的） 和 applicative（含有ap方法的）：

```
// 从 chain 衍生出的 map
X.prototype.map = function(f) {
  var m = this;
  return m.chain(function(a) {
    return m.constructor.of(f(a));
  });
}

// 从 chain/map 衍生出的 ap
X.prototype.ap = function(other) {
  return this.chain(function(f) {
    return other.map(f);
  });
};
```

这一点非常强大，甚至可以审查一个数据类型，然后自动化这个过程。处理多个仿函数作为参数的情况，是 applicative functor 一个非常好的应用场景。借助applicative，能够在仿函数的世界里调用函数。尽管已经可以通过 monad 达到这个目的，向下的嵌套结构使得 monad 拥有串行计算、变量赋值和暂缓后续执行等独特的能力，但在不需要 monad 特定功能的时候，最好使用 applicative，用合适的方法来解决合适的问题。

给个最常见的例子，假设要创建一个旅游网站，既需要获取游客目的地的列表，还需要获取地方事件的列表。这两个请求仅仅需要通过相互独立的 api 调用。

```
// Http.get :: String -> Task Error HTML

var renderPage = curry(function(destinations, events) { /* render page */  });

Task.of(renderPage).ap(Http.get('/destinations')).ap(Http.get('/events'))
// Task("<div>some page with dest and events</div>")
```

两个请求将会同时立即执行，当两者的响应都返回之后， `renderPage` 就会被调用。这与 monad 版本的那种必须等待前一个任务完成才能继续执行后面的操作完全不同。本来就无需根据目的地来获取事件，因此也就不需要依赖顺序执行。由此来看 applicative 处理异步和并发是多么简单的事情。还有更多更好的实例，请参阅相关文档。

总结

本文作为函数式编程的入门文章，主要面向初学者，以及习惯了面向对象编程的程序员。文章从思维和实践角度来研究函数式编程方法，针对编码中的常见问题列举了对应的基本解决思路，一些概念与其他编码方法做了类比，可能并不十分确切，请阅读时区分辨别。因为文章重在思维逻辑，并没罗列函数式编程的过多内容和方式方法，提到的内容也没有做深入探讨，参考里列举了一些优秀文档，含有大量分析解读，请自行查阅。

目前，开源社区提供了很多优秀的函数式编程库，比如：lodash/FP，ramda.js，Folktale等，文章 Javascript的函数式库 还列出了其他一些优秀的库，并作了深入分析和介绍。这些库，既可以在生产中使用，更可以作为学习研究的范本，值得学习参考。亿书将在未来的版本和产品中，认真学习实践这些优秀的编程方法，进一步减少核心库的代码量，提高主链和侧链的综合性能。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

亿书白皮书： http://ebookchain.org/ebookchain.pdf

亿书官网： http://ebookchain.org

亿书官方QQ群：185046161（亿书完全开源开放，欢迎各界小伙伴参与）

参考

lodash FP Guide

ramda.js

Folktale

mostly-adequate-guide

函数式编程初探

使用 JavaScript 进行函数式编程

结构化编程

Functional programming

函数式编程（百度百科）

柯里化（维基百科）

Tacit programming

Hindley–Milner type system

Functor, Applicative, 以及 Monad 的图片阐释

图解 Monad

Functor（维基百科）

JavaScript Function Composition

Javascript的函数式库

轻松从Js文件生成UML类图

前言

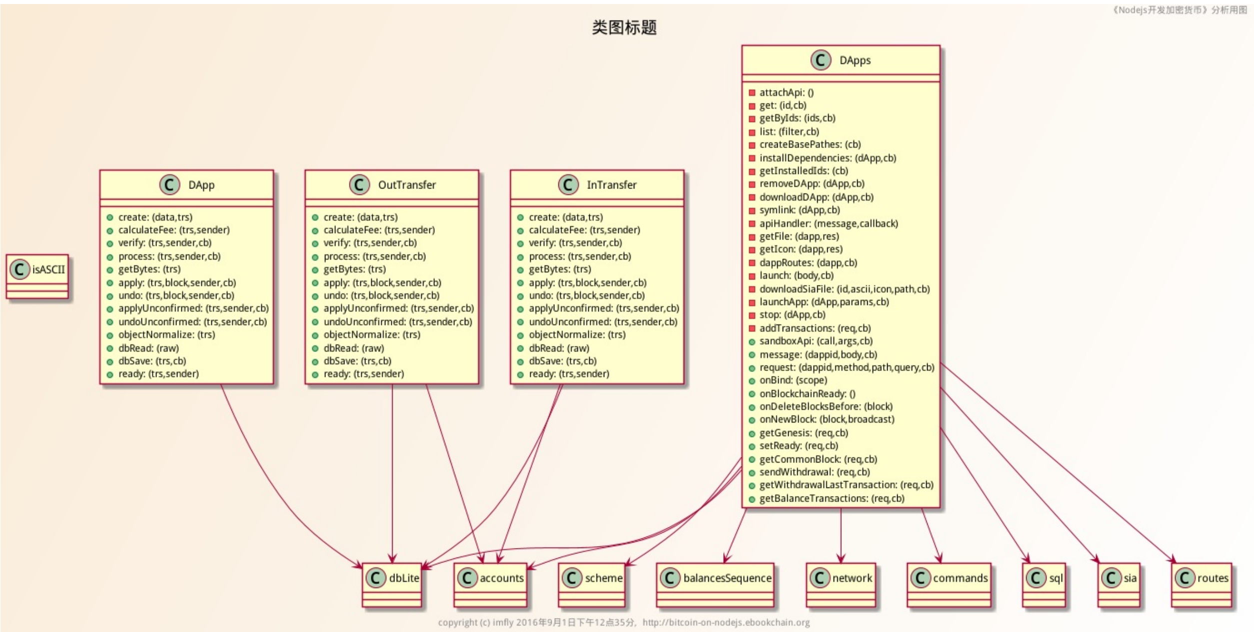
上一篇《函数式编程入门经典》，罗嗦了很长，很多小伙伴看得云里雾里。这里提供一个实例，让大家切身感受函数式编程的奥妙和趣味。当然，仅仅为了举例而写代码就没有什么意义了，本书提供的例子都是承担了某项任务的具体项目或工具，这个例子自然也不能例外。

本书用到了大量的Uml类图，经常有小伙伴问我用什么工具画的。说实话，前几篇是我个人一点点手工整理的，但后来就感觉在浪费生命，作为程序员，怎么可能容忍这样的事情反复发生。所以，就有了 js2uml（见参考）这个小工具。只不过，当初目的单一，仅仅使用正则表达式过滤js代码（v0.1.0），所以不够灵活，自然也没有单独放出来。现在引入了抽象语法树，顺带使用函数式编程进行了重构，也让该工具更加通用了。

因为代码量很少，并且前面对函数式编程进行了大量阐述，这里就不再详细描述编码过程了，仅仅把设计思路和流程提供出来，喜欢的小伙伴请自己查阅源代码吧。

工具简要介绍

这是一个命令行工具，可以轻松从Js文件生成UML类图，比如：



主要实现了以下两个功能：

（1）直接读取js源码，产生 plantuml 认识的格式文件。我个人喜欢用 `.pu` 作为后缀。内容格式，比如：

```
@startuml
title 区块链相关类图及其关联关系
footer copyright (c) imfly 2016.07.23 http://bitcoin-on-nodejs.ebookchain.org
header 《Nodejs开发加密货币》分析用图：《区块链》

  Class Loader {
    ...
  }

  ' relationship
  Block -right-> transaction
@enduml
```

然后，使用 Graphviz 工具就可以导出png等格式的图片了。之所以提供导出这类格式的文件，主要是工具对于排版的处理还不够智能，布局、关联关系、背景图片等，有时需要通过手工进一步修改。另外，这也为以后与其他工具的集成，提供了便利。

（2）直接生成png,svg等格式的图片文件。这是对 Graphviz 的直接扩展。

具体安装使用方法非常简单，请直接查看它的文档吧，这里不再赘述。

能从中学点什么？

通过这个小工具的开发，可以学习到下面的技能：

- 抽象语法树（Abstract Syntax Tree, AST）的处理。抽象语法树有很多用途，大部分人可能很少直接在代码里使用，但是几乎每个人也都在用，比如：编辑器的自动提示、自动完成等功能，使用Nodejs的小伙伴，最后都要对代码进行混淆、压缩等处理，那些工具也都要用到；
- 函数式编程。原来的版本，是完全面向对象的编程方式，也没有使用AST，所以代码长，功能有限。这个版本，进行了优化，使用函数式编程，大大缩减了代码量，还提供了直接导出.png/.svg/.pu等各种格式图片的能力;
- 学习plantuml。这是使用代码处理Uml的最好方式，直接使用dot语言（生成的.pu文件就是），像编写程序一样画Uml图，真的只有程序猿才能体会的畅快。
- 命令行工具开发。

关于抽象语法树

在没有任何运行环境的情况下，要想对源码进行分析，通常有两种方法可选，一个就是正则表达式，正如第一版本里用到的，但是正则表达式仅能抽取源码有限的格式，很容易把格式固定，特别是对js这种没有真正的类的概念的脚本语言来说，就更不通用。抽象语法树是程序

的一种中间表示形式，是专门用于程序分析的。凡是涉及到对源程序进行操作和处理的应用，都会用到抽象语法树，比如我们大家经常使用的智能编辑器、语言翻译器等。因此，更好的，也是最通用方法就是使用抽象语法树。

抽象语法树的具有不依赖于具体编程语言文法和语言细节的特点，所以对于不同的编码方式，甚至不同的编程语言，在语法分析的时候，都能构造出相同的语法树，这就为后端实现了清晰，统一的接口。这对于把控代码质量，控制编码行为，甚至必要的时候批量重构，提供了可行的操作方法。我们之所以要研究和使用这项技术，一个根本目的是一个与亿书项目配套的辅助项目——亿书的远程开源协作开发平台。

该平台，能监控每一个贡献者的代码数量和质量，给出优化建议，对贡献者的每次代码贡献，通过亿书智能合约自动支付合理的亿书币报酬，最后写入区块链。这个平台将改变程序员的生活方式，让按照代码贡献计酬更加科学，加之区块链智能合约的底层支持，程序员无论身处何处，都可以随时贡献自己的智慧，并获得收益，让远程办公成为非常简单和现实的事情。

前面也说了，对js代码的混淆和压缩，都用到了抽象语法树。著名的混淆和压缩工具 UglifyJS 最早是自己开发的解析工具，但是最新版本 UglifyJS2 使用acorn进行了重写。在Node.js的世界里，有几个比较优秀的抽象语法树处理包，除了acorn，还有一个Esprima。我这里使用的是Esprima。

工具实现过程

（1）基本需求

这个工具需求非常简单，就是给一个js源码文件，直接导出分析后的Uml文件或图片。这类简单的应用模型，非常普遍，甚至可以说任何大的应用项目，都是由这类简单的应用模型搭积木似的搭起来的。我们在入门部分提到过，使用NOde.js，最好习惯数据“流”的概念，这里从js文件到Uml或图片文件的过程，就是典型的数据“流”的实现。我们要做的，仅仅是在这个“流”上制造一些“过滤器”而已。

上一篇关于函数式编程的文章里，我们也说函数式编程更适合处理数据“流”，其做法的一个重点是告诉程序“是什么”（声明式），而不是“怎么干”（命令式）。换句话说，我们只要好好描述这里的“过滤器”是什么就行了，千万不要关心它们怎么干。比如开发的过程中，为了调试方便，我先是这么搭积木的（命令式）：

```
// 文件 ./lib/index.js
// 26行
function(data) {
    var functionList = parser(data);
    var formatData = format(functionList);
    var result = template(formatData);

    ...
}
```

这段代码一看就可以写成这样：

```
var result = template(format(parser(data)));
```

所以，这段代码在函数式编程里就是（我们使用ramda来处理函数式编程）：

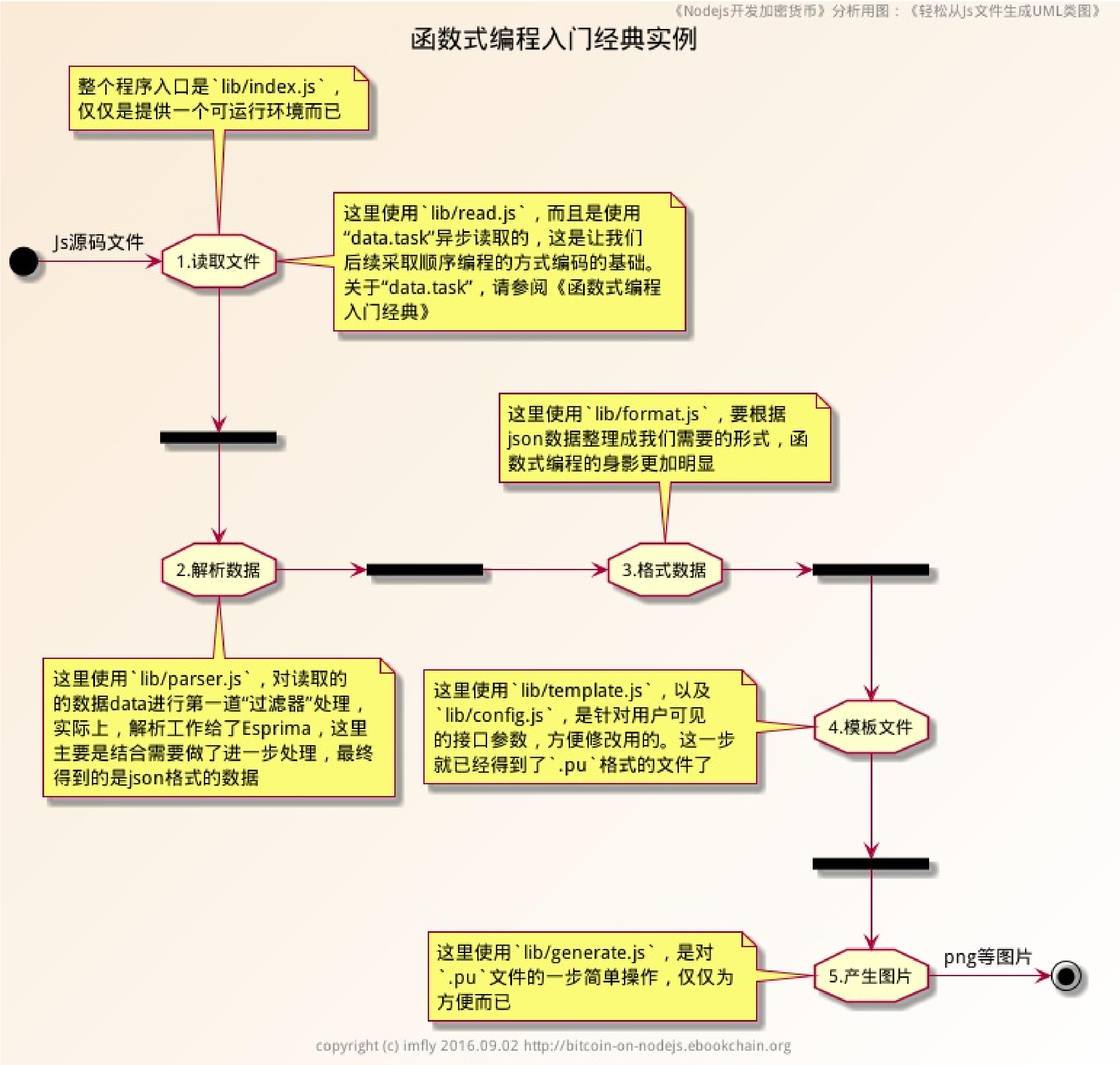
```
// 11行
var getPuml = _.compose(template, format, parser); //声明式

// 28行
var puml = getPuml(data);
```

沿着这个思路逐个拆分和组合，就非常清晰的处理了整个过程。

（2）架构流程和代码结构

还是用一张图来说明吧，非常简单。



总结

关于编码的方法论，这两篇文章已经足够。这篇文章仅仅是为函数式编程提供了一个小例子。例子本身并不完美，没有收集属性变量，也没有添加测试（实际项目不提倡），很多函数式编程的高级特性也都没有涉及到，不能算是完全的函数式编程，后续会进一步完善提升。在亿书后续的开发中，也会使用大量函数式编程的好经验、好做法，进一步压缩代码量，提高健壮性和可维护性。

凡事过犹不及，尽管用了大量篇幅来介绍函数式编程，但是并不代表要大家完全使用函数式编程来开发，如果能让大家在处理一些问题的时候，不自觉的使用一些函数式编程的方法实践，也就达到目的了。比如根据需要，对某个功能、模块，甚至仅仅是某些函数，适当合理的采用函数式编码的方式，已经足够提高工作效率和代码健壮性了。

链接

本系列文章即时更新，若要掌握最新内容，请关注下面的链接

本源文地址： https://github.com/imfly/bitcoin-on-nodejs

亿书官网： http://ebookchain.org

亿书官方QQ群：185046161（亿书完全开源开放，欢迎各界小伙伴参与）

参考

js2uml

v0.1.0

plantuml

Abstract syntax tree

Fun with Esprima and Static Analysis

UglifyJS

UglifyJS2

区块链相关术语（中英对照）

说明：阅读英文文档是编程开发过程中最常做的一件事，英文阅读也是一个程序员的基本能力。区块链刚刚起步，每天各种新概念层出不穷，为方便大家学习和使用，这里收录了巴比特论坛上的一个帖子内容。该帖子仍在持续更新，更多新内容请点击下面的地址阅读原帖。

原文标题：《数字货币翻译术语（中英对照）》

原文地址：http://8btc.com/thread-17286-16-1.html

English	中文
account level(multiaccountstructure)	账户等级（多账户结构）
accounts	账户
adding blocks to	增加区块至
addition operator	加法操作符
addr message	地址消息
Advanced Encryption Standard(AES)	高级加密标准(AES)
aggregating	聚合
aggregating into blocks	聚集至区块
alert messages	警告信息
altchains	竞争币区块链
altcoins	竞争币
AML	反洗钱
anonymity focused	匿名的
antshares	小蚁
appcoins	应用币
API	应用程序接口
App Coins	应用币
architecture	架构
assembling	集合
attacks	攻击
attack vectors	攻击向量

Autonomous Decentralized Peer-to-Peer Telemetry	去中心化的 p2p 自动遥测系统
auxiliary blockchain	辅链
authentication path	认证路径
B	
backing up	备份
balanced trees	平衡树
balances	余额
bandwidth	带宽
Base58 Check encoding	Base58Check编码
Base58 encoding	Base58编码
Base-64 representation	Base-64表示
BFT（Byzantine Fault Tolerance）	拜占庭容错
binary hash tree	二叉哈希树
BIP0038 encryption	BIP0038加密标准
bitcoin addressesvs.	比特币地址与
bitcoin core engine	比特币核心引擎或网络
bitcoin ledger	比特币账目
bitcoin network	比特币网络
Bitcoin Network Deficit	比特币网络赤字
Bitcoin Miners	比特币矿工
Bitcoin mixing services	混币服务
Bitcoin source code	比特币源码
BitLicense	数字货币许可
BIP152	比特币改进提议
Bitmain	比特大陆
Bitmessage	比特信
BITNET	币联网
Bitshares	比特股
BitTorrent	文件分享
Blake algorithm	Blake算法
block chain apps	区块链应用

block generation rate	出块速度
block hash	区块散列值
block header hash	区块头散列值
block headers	区块头
block height	区块高度
blockmeta	区块元
block templates	区块模板
blockchains	区块链
bloom filtersand	布鲁姆过滤器(bloom过滤器)
BOINC open grid computing	BOINC开放式网格计算
brain wallet	脑钱包
broad casting to network	全网广播
broad casting transactions to	广播交易到
bytes	字节
Byzantine fault-tolerant	拜占庭容错
C	
call	调用
CCVM（Cross Chain Virtual Machine）	跨链交易的虚拟机
centralized control	中心化控制
chaining transactions	交易链条
chainwork	区块链上工作量总值
Check Block function(Bitcoin Coreclient)	区块检查功能(BitcoinCore客户端)
CHECKMULTISIG implementation	CHECKMULTISIG实现
CheckSequenceVerify (CSV)	检查序列验证/CSV
checksum	校验和
child key derivation(CKD) function	子密钥导出(CKD)函数
child private keys	子私钥
Child Pays For Parent，CPFP	父子支付方案
coinbase reward calculating	coinbase奖励计算
coinbase rewards	coinbase奖励
coinbase transaction	coinbase交易
cold-storage wallets	冷钱包

Compact block	致密区块
Compact block relay	致密区块中继
colored coins	彩色币
compressed keys	压缩钥
compressed private keys	压缩格式私钥
compressed public keys	压缩格式公钥
computing power	算力
connections	连接
consensus	共识
Consensus Ledger	共识账本
consensus attacks	一致性功能攻击
consensus innovation	一致性的创新
consensus plugin	共识算法
Confidential Transactions	保密交易
constant	常数
constructing	建造
constructing block headers with	通过...构造区块头部
converting compressed keys to	将压缩地址转换为
converting to bitcoin addresses	转换为比特币地址
conversion fee	兑换费用
consortium blockchains	共同体区块链
counterparty protocol	合约方协议
Counterparty	合约币
creating full blockchains on	建立全节点于
creating on nodes	在节点上新建
crypto community	加密社区
crypto 2.0 ecosystem	加密2.0生态系统
cryptocurrency	加密货币
Cunning hamprime chains	坎宁安素数链
currency creation	货币创造
D	

Darkcoin	暗黑币（译者注：现已更名为达世币Dash）
data structure	数据结构
DAO(Decentralized Autonomous Organization)	去中心化自治组织
Debt Token	债权代币
decentralized	去中心化
decentralized consensus	去中心化共识
decentralised applications	去中心化应用
decentralised platform	去中心化平台
decoding Base58Check to/from hex	Base58Check编码与16进制的相互转换
decoding to hex	解码为16进制
deep web	深网
Decode Raw Transaction	解码原始交易
deflationary money	通缩货币
delegated proof of stake	授权股权证明机制
demurrage currency	滞期费
denial of service attack	拒绝服务攻击
detached block	分离块
deterministic wallets	确定性钱包
DEX ：distributed exchange	去中心化交易所
difficulty bits	难度位
difficulty retargeting	难度调整
difficulty targets	难度目标
digital notary services	数字公正服务
digital currency	数字货币
distributed hash table	分布式哈希表
Distributed Autonomous Corporations Runtime System（DACRS）	自治系统运行环境
Distributed Ledger Technology（DLT）	分布式账簿技术
domain name service(DNS)	域名服务(DNS)
double-spend attack	双重支付攻击
double spend	双花

Dogecoin	狗狗币
DoS(denial of service) attack	拒绝服务攻击
DPOS	权益代表证明机制/DPOS算法（POS基础上的改良）
dual-purpose	双重目标
dual-purpose mining	双重目的挖矿
dust rule	尘额规则（极其小的余额）
E	
eavesdroppers	窃听者
ecommerce servers keys for…	电子商务服务器...的密钥
ECDSA	椭圆曲线数字签名算法保障
Eigentrust++ for nodes	用于节点的Eigentrust++技术
electricity cost	电力成本
electricity cost and target difficulty	电力消耗与目标难度
Electrum wallet	Electrum 钱包
ellipticcurve multiplication	椭圆曲线乘法
Emercoin(EMC)	崛起币
encoding/decoding from Base58Check	依据Base58Check编码/解码
encrypted	加密
encrypted private keys	加密私钥
Equity Token	权益代币
Ethereum	以太坊
External owned account（EOA）	外有账户
ether	以太币
extended key	扩展密钥
extra nonce solutions	添加额外nonce的方式
extraBalance	附加余额
F	
Factom	公证通
fault tolerance	外加容错
Feathercoin	羽毛币
fees	手续费

FRN	快速中继网络
FBRP	快速区块中继协议
FEC	向前纠错
field programma blegatearray(FPGA)	现场可编程门阵列(FPGA)
Financial disintermediation	金融脱媒
fintech	金融技术
fork attack	分叉攻击
forks	分叉
fraud proofs	欺诈证明
full nodes	完整节点;全节点
G	
generating	生成
generation transaction	区块创始交易
generator point	生成点
genesis block	创始区块
GetBlock Template(GBT)mining protocol	GetBlockTemplate(GBT)挖矿协议
gettingon SPV nodes	获取SPV节点
GetWork(GWK) mining protocol	GetWork(GWK)挖矿协议
graphical processing units(GPUs)	图形处理单元(GPUs)
GUID	全域唯一识别元
H	
hackers	黑客
halving	减半
hardware wallets	硬件钱包
hard fork	硬分叉
hard limit	硬限制
hash	哈希值
Hardware Security Modules（HSM）	硬件安全模块
hashing powerand	哈希算力
hashcash	现金算法
HD wallet system	分层确定性钱包系统
header hash	头部散列值

heavyweight wallet	重量级钱包
Hierarchy deterministic	分层确定的
honesty	诚信算力
hyperledger	超级账本
human readable format	人类可读模式
I	
identifiers	标识符
immutability of blockchai	区块链不可更改性
implementing in Python	由Python实现
in block header	在区块的头部
independent verificatio	独立验证
innovation	创新
inputs	输入
Internet of Things	物联网
instamine	偷挖
Invertible Bloom Lookup Table(IBLT)	可逆式布鲁姆查找表
Invalid Numerical Value	无效数值
IPDB	星际数据库
K	
key formats	密钥格式
key-value	键值
KYC	了解你的客户
L	
Level DB database(Google)	LevelDB数据库(Google)
light weight	轻量级
linking blocks to…	将区块连接至…
linking to blockchain	连接至区块链
Lightning network	闪电网络
linear scale	线性尺度
Litecoin	莱特币
lock time	锁定时间

locking scripts	锁定脚本
log scale	对数单位
M	
mainnet	主网
managed pools	托管池
mastercoin protocol	万事达币协议
masternode	主节点
memorypool(mempool)	内存池
Merkle tree(Merkle Hash tree)	二进制的哈希树或者二叉哈希树
Merkle root	二进制哈希树根
metachains	附生块链
mining	挖矿
mining blocks successfully	成功产(挖)出区块
mining pools	矿池
mining rigs	矿机
micropayment	小额支付
microblocks	微区块
modifying private key formats	修改密钥格式
monetary parameter alternatives	货币参数替代物
Moore’s Law	摩尔定律
Moonpledge	月球之誓
MPC	多方计算
multi account structure	多重账户结构
multi-hop network	多跳网络
multi-signature	多重签名
multi-signature addresse	多重签名地址
multi-signature addresses	多重签名地址
multi-signature scripts	多重签名脚本
multi-signatureaccount	多重签名账户
N	
Namecoin	域名币
native token	原生代币

navigating	导航
Network Propagation	网络传播算法
Network of marketplaces	市场网络
Nextcoin（NXT)	未来币
Neoscrypt	N算法
nested subchains	嵌套子链
NFC(Near Field Communication)	非接触式
NIST5	NIST5是一种新算法，由TalkCoin首创
nodes	节点
nonce	随机数
noncurrency	非货币
nondeterministic wallets	非确定性的
O	
off-chain	链下
on full nodes	在全节点上
on new nodes	在新节点上
on SPV nodes	在SPV节点
on the bitcoin network	在比特币网络中
one-hop network	单跳网络
OP_RETURN operator	OP_RETURN操作符
OpenSSL cryptographiclibrary	OpenSSL密码库
open source of bitcoin	比特币的开源性
open transaction（OT)	开放交易
orphan block	孤儿块
Oracles	价值中介
OWAS	单向聚合签名
OTC(over the counter)	场外交易
outputs	输出
P	
P2P Pool	P2Pool（一种点对点方式的矿池）
parent blocks	父区块

parent blockchain	主链
paths for	路径
Pay to script hash (P2SH)	P2SH代码；脚本哈希支付方式
payment channel	支付通道
P2SH address	P2SH地址；脚本哈希支付地址
peer-to-peer networks	P2P网络
physical bitcoin storage	比特币物理存储
PIN-verification	芯片密码
plot/chunks of data	完整数据块
pool operator of mining pools	矿池运营方
post-trade	交易后
post-trade processing	交易后处理
POI: proof of importance	重要性证明（NEM提出来的一种共识算法）
Ppcoin	点点币
Premine	预挖
priority of transactions	交易优先级
Primecoin	素数币
proof of stake	权益证明
proof of work	工作量证明
proof-of-work algorithm	工作量证明算法
proof-of-work chain	工作量证明链
propagating transactions on	交易广播
protein folding algorithms	蛋白质折叠算法
public child key derivation	公钥子钥派生
public key derivation	公钥推导
publickeys	公钥
public blockchain/permissionless blockchain	公链
private blockchain/permissioned blockchain	私链
pump and bump	拉升出货
purpose level(multiaccount structure)	目标层（多帐户结构）
Python ECDSA library	PythonECDSA库

R	
random	随机
random wallets	随机钱包
raw value	原始价格
reentrancy	可重入性
regtech	监管技术
replay attacks	重放攻击
RBF:Replace By Fee	费用替代方案
retargeting	切换目标
recursive call	递归调用
RIPEMD160	RIPEMD160一种算法
Ripple	瑞波币
risk balancing	适度安保
risk diversifying	分散风险
root of trust	可信根
root seeds	根种子
S	
sandbox	沙箱
satoshis	中本聪
scoops/4096 portions	子数据块
scriptcons truction	脚本构建
scriptl anguage for	脚本语言
Scriptlanguage	脚本语言
scripts	脚本
scrypt algorithm	scrypt算法
scrypt-N algorithm	scrypt-N算法
Secure Hash Algorithm(SHA)	安全散列算法
security	安全
security thresholds	安全阈值
seed nodes	种子节点
seeded	种子
seeded wallets	种子钱包

selecting	选择
soft limit	软限制
Segregated Witness(SegWit)	隔离见证
SHA256	SHA256
SHA3 algorithm	SHA3算法
Shared Permission Blockchain	共享认证型区块链
shibes	狗狗币粉丝
shopping carts public keys	购物车公钥
simplified payment verification (SPV) nodes	简易支付验证（SPV）节点
simplified payment verification（SPV）wallet	轻钱包
sidechain	侧链
signature operations（sigops）	处理签名操作
signature aggregation	签名集合
Skein algorithm	Skein算法
smart pool	机枪池
smart contracts	智能合约
solo mining	单机挖矿
solo miners	独立矿工
soft fork	软分叉
spilt	分割
Stellar	恒星币
stateless verification of transactions	交易状态验证
statelessness	无状态
state machine replication	状态机原理
storage	存储
Stratum(STM)mining protocol	Stratum挖矿协议
structure of	的结构
sx tools	sx工具
syncing the blockchain	同步区块链
system security	系统安全
Subchains	子链

T	
taking off blockchain	从区块链中删除
tainted address	被污染的地址
taint Analysis	污点分析
TeleHash	p2p信息发送系统
timeline	时间轴
timestamping blocks	带时间戳的区块
txids	缩短交易标识符
token	代币
token system	代币系统
token-less blockchain	无代币区块链=私链
transaction fees	交易费；矿工费
transaction pools	交易池
transaction processing	交易处理
transaction validation	交易验证
transactions independent verification	独立验证交易
transaction malleability	交易延展性
tree structure	树结构
Trezor wallet	Trezor钱包
Turing Complete	图灵完备
two-factor authentication	双因素认证
tx messages	tx消息
Type-0 nondeterministic wallet	原始随机钱包
U	
uncompressed keys	解密钥
unconfirmed transactions	未确认交易
unspent outputs	未花费输出
user security	用户安全性
User Token	用户代币
UTXO pool	UTXO池
UTXO set	UTXO集合
UTXOs	未交易输出

V	
validating new blocks	验证新区块
validation	验证条件
validation(transaction)	校验(交易)
vanity	靓号
vanity addresses	靓号地址
vanity-miners	靓号挖掘程序
verification	验证
verification criteria	验证条件
version message	版本信息
Visualise Transaction	可视化交易
W	
Wallet Import Format(WIF)	钱包导入格
wallets	钱包
white hat attack	白帽攻击
weak blocks	弱区块
whitelist	白名单
wildcard	通配符
X	
Xthin	极瘦区块
XRP (Ripple)	瑞波币
Z	
zero knowledge proof	零知识证明
zero codehash	零代码哈希
Zerocoin protocol	零币协议

区块链相关名词汇总

前言

新技术的最大问题是新概念太多。例如加密的概念，包括散列、签名、公钥、私钥、对称和不对称加密，还有分布式哈希表、信任网络等。还有额外的内部术语，如“区块”，“确认”，“挖矿”，“SPV客户”和“51％攻击”等，这通常是学习掌握新技术过程中必须了解的。为了方便，在社区小伙伴`珍惜`、`一`、`Tailor`、`Mojie`等帮助下，通过网络收集，并进行了适当删改整理，供大家在学习乃至工作过程中参考使用。

密码学

计算上不可行：一个处理被称为是计算上不可行，如果有人想有兴趣完成一个处理但是需要采取一种不切实际的长的时间来做到这一点的（如几十亿年）。通常，2的80次方的计算步骤被认为是计算上不可行的下限。

散列：一个散列函数（或散列算法）是一个处理，依靠这个处理，一个文档（比如一个数据块或文件）被加工成看起来完全是随机的小片数据（通常为32个字节），从中没有意义的数据可以被复原为文档，并且最重要的性能是散列一个特定的文档的结果总是一样的。

此外，极为重要的是，找到具有相同散列的两个文件在计算上是不可能的。一般情况下，即使改变文件的一个字母也将完全打乱散列;例如，“ Saturday”的SHA3散列为c38bbc8e93c09f6ed3fe39b5135da91ad1a99d397ef16948606cdcbd14929f9d，而Caturday的SHA3散列是b4013c0eed56d5a0b448b02ec1d10dd18c1b3832068fbbdc65b98fa9b14b6dbf。散列值经常被用作以下用途：为无法伪造的特定文档而创建的全局商定标识符。

加密：与被称为钥匙（例如c85ef7d79691fe79573b1a7064c19c1a9819ebdbd1faaab1a8ec92344438aaf4）的短字符串的数据相结合，对文档（明文）所进行的处理。加密会产生一个输出（密文），这个密文可以被其他掌握这个钥匙的人“解密”回原来的明文，但是对于没有掌握钥匙的人来说是解密是费解的且计算上不可行。

公钥加密：一种特殊的加密，具有在同一时间生成两个密钥的处理（通常称为私钥和公钥），使得利用一个钥匙对文档进行加密后，可以用另外一个钥匙进行解密。一般地，正如其名字所建议的，个人发布他们的公钥，并给自己保留私钥。

数字签名：数字签名算法是一种用户可以用私钥为文档产生一段叫做签名的短字符串数据的处理，以至于任何拥有相应公钥，签名和文档的人可以验证（1）该文件是由特定的私钥的拥有者“签名”的，（2）该文档在签名后没有被改变过。请注意，这不同于传统的签名，在传统签名上你可以在签名后涂抹多余的文字，而且这样做无法被分辨;在数字签名后任何对文档的改变会使签名无效。

区块链

地址：一个地址本质上是属于特定用户的公钥的表现;例如，与上面给出的私钥的相关联的地址是cd2a3d9f938e13cd947ec05abc7fe734df8dd826。注意，在实际中，地址从技术上来说是一个公钥的散列值，但为了简单起见，最好忽略这种区别。

交易：一个交易是一个文档，授权与区块链相关的一些特定的动作。在一种货币里，主要的交易类型是发送的货币单位或代币给别人;在其他系统，如域名注册，作出和完成报价和订立合约的行为也是有效的交易类型。

区块：一个区块是一个数据包，其中包含零个或多个交易，前块（“父块”）的散列值，以及可选的其它数据。除了初始的“创世区块”以外每个区块都包含它父块的散列值，区块的全部集合被称为区块链，并且包含了一个网络里的全部交易历史。注意有些基于区块链的加密货币使用“总账”这个词语来代替区块链。这2者的意思是大致相同的，虽然在使用“总账”这个术语的系统里，每个区块都通常包括每个账户的目前状态（比如货币余额，部分履行的合约，注册）的全部拷贝，并允许用户抛弃过时的历史数据。

创世区块:创世区块指区块链上的第一个区块，用来初始化相应的加密货币。

帐户：帐户是在总账中的记录，由它的地址来索引，总账包含有关该帐户的状态的完整的数据。在一个货币系统里，这包含了货币余额，或许未完成的的交易订单;在其它情况下更复杂的关系可以被存储到账户内。

随机数：在一个区块里的一个无意义的值，为了努力满足工作证明的条件来进行调整。

挖矿：挖矿是反复总计交易，构建区块，并尝试不同的随机数，直到找到一个随机数可以符合工作证明的条件的过程。如果一个矿工走运并产生一个有效的区块的话，会被授予的一定数量的币（区块中的交易全部费用）作为奖励。而且所有的矿工开始尝试创建新的区块，这个新区块包含作为父块的最新的区块的散列。

陈腐区块：对于同一个父块，已经有另外一个区块被创建出来之后，又被创建的区块;陈旧区块通常被丢弃，是精力的浪费。

幽灵（**Ghost**）：幽灵是一个协议，通过这个协议，区块可以包含不只是他们父块的散列值，也散列父块的父块的其他子块（被称为叔块）的陈腐区块。这确保了陈腐区块仍然有助于区块链的安全性，并减轻了大型矿工在快速区块链上的有优势的问题，因为他们能够立即得知自己的区块，因此不太可能产生陈腐区块。

叔块：是父区块的父区块的子区块，但不是自个的父区块，或更一般的说是祖先的子区块，但不是自己的祖先。如果A是B的一个叔区块，那B是A的侄区块。

分叉：指向同一个父块的2个区块被同时生成的情况，某些部分的矿工看到其中一个区块，其他的矿工则看到另外一个区块。这导致2种区块链同时增长。通常来说，随着在一个链上的矿工得到幸运并且那条链增长的话，所有的矿工都会转到那条链上，数学上分几乎会在4个区块内完结自己。

硬分叉，是当比特币协议规则发生改变，旧节点拒绝接受由新节点创造的区块的情况。违反规则的区块将被忽视，矿工会按照他们的规则集，在他们最后见证的区块之后创建区块。

软分叉，是当比特币协议规则发生改变，旧的节点并不会意识到规则是不同的，它们将遵循改变后的规则集，继续接受由新节点创造的区块。矿工们可能会在他们完全没有理解，或者验证过的区块上进行工作。

双重花费：是一个故意的分叉，当一个有着大量挖矿能力的用户发送一个交易来购买产品，在收到产品后又做出另外一个交易把相同量的币发给自己。攻击者创造一个区块，这个区块和包含原始交易的区块在同一个层次上，但是包含并非原始交易而是第二个交易，并且开始在这个分叉上开始挖矿。如果攻击者有超过50％的挖矿能力的话，双重花费最终可以在保证在任何区块深度上成功。低于50％的话，有部分可能性成功。但是它经常在深度2-5上有唯一显著的可能。因此在大多数的加密货币交易所，博彩站点还有金融服务在接受支付之前需要等待6个区块被生产出来（也叫“6次确认”）。

比特币等区块链产品

BIP：比特币改进提议（Bitcoin Improvement Proposals 的缩写），指比特币社区成员所提交的一系列改进比特币的提议。例如，BIP0021是一项改进比特币统一资源标识符（URI）计划的提议。

比特币：“比特币”既可以指这种虚拟货币单位，也指比特币网络或者网络节点使用的比特币软件。

确认：当一项交易被区块收录时，我们可以说它有一次确认。矿工们在此区块之后每再产生一个区块，此项交易的确认数就再加一。当确认数达到六及以上时，通常认为这笔交易比较安全并难以逆转。

难度：整个网络会通过调整“难度”这个变量来控制生成工作量证明所需要的计算力。

难度目标：使整个网络的计算力大致每10分钟产生一个区块所需要的难度数值即为难度目标。

难度调整：BTC整个网络每产生2,106个区块后会根据之前2,106个区块的算力进行难度调整。

矿工费：交易的发起者通常会向网络缴纳一笔矿工费，用以处理这笔交易。大多数的交易需要0.5毫比特币的矿工费。

哈希：二进制数据的一种数字指纹。

矿工：矿工指通过不断重复哈希运算来产生工作量证明的各网络节点。

网络：比特币网络是一个由若干节点组成的用以广播交易信息和数据区块的P2P网络。

奖励：每一个新区块中都有一定量新创造的比特币用来奖励算出工作量证明的矿工。现阶段每一区块有25比特币的奖励。

私钥：用来解锁对应（钱包）地址的一串字符，例如5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh。

交易：简单地说，交易指把比特币从一个地址转到另一个地址。更准确地说，一笔"交易"指一个经过签名运算的，表达价值转移的数据结构。每一笔"交易"都经过比特币网络传输，由矿工节点收集并封包至区块中，永久保存在区块链某处。

钱包：钱包指保存比特币地址和私钥的软件，可以用它来接受、发送、储存你的比特币。

SPV客户端（或轻客户端）：一个只下载一小部分区块链的客户端，使拥有像智能手机和笔记本电脑之类的低功率或低存储硬件的用户能够保持几乎相同的安全保证，这是通过有时选择性的下载的小部分的状态，而在区块链验证和维护时，不需要花费兆字节的带宽或者千兆字节的存储空间。

楔入式侧链技术（ **pegged sidechains**）：它将实现比特币和其他数字资产在多个区块链间的转移，这就意味着用户们在使用他们已有资产的情况下，就可以访问新的加密货币系统。

工作量证明（**Proof-of-Work**）:一种共识机制，该机制是一方（通常称为证明人）出示计算结果，这个结果众所周知是很难计算的但却很容易验证的。通过验证这个结果，任何人都能够确认证明人执行了一定量的计算工作量来产生这个结果。

权益证明（**Proof of Stake**）: 一种共识机制，该机制是当创造一个区块时，矿工需要创建一个"币权"交易，交易会按设定的比例把一些币发送给矿工本身，类似利息。

股份授权证明机制（**DPOS**）: 一种共识机制，该机制让每一个持有币的人对整个系统资源当代表的人进行投票，而获得最多票数的101个代表获得进行交易打包计算的权利，而系统给予对应的奖励。

RChain：是具有并发和分布式的区块链。"分布式"指的是区块链细分成组合件，它连成一个统一的整体，而不需要一次性全部计算（而比特币区块链则需要）。"并发"的意思是，这个分支使不同的进程能够平行运行，而且不会互相干扰。

Rholang：是RChain的本土智能合约语言（或编程语言），一种反射性的、高阶过程编程语言，基于进程演算，允许进程的并行执行和在低阶智能合约基础上组合高阶智能合约，以一种高效和安全的方式，允许在正常的验证基础上进行更好的安全性测试和模拟

SpecialK：一种分布式存储技术的方法，它提供了一个单一领域的特定语言，为程序员们提供了一个熟悉的、统一的API，通过API他们可以访问分布在整个网络的数据。数据被分配时始终兼顾冗余度和敏感度，确保随时随地有需求时它都是可用的，并且不需要时就会隐藏。

零知识证明: 证明者和验证者之间进行交互，证明者能够在不向验证者提供任何有用的信息的情况下，使验证者相信某个论断是正确的。

比特币的可替换性(**Fungibitlity**）：持有的比特币不管之前曾进行过哪些交易历史，包括可能涉及过毒品交易等，这都与刚挖出来的“原币”一样，完全可以平等替换。现在有交易所或其他服务公司会追踪用户账户比特币的来源，一旦涉及犯罪，他们会不接受。

环签名：因签名中参数Ci(i=1,2,…,n)根据一定的规则首尾相接组成环状而得名。其实就是实际的签名者用其他可能签字者的公钥产生一个带有断口的环，然后用私钥将断口连成一个完整的环。任何验证人利用环成员的公钥都可以验证一个环签名是否由某个可能的签名人生成。

隔离见证：Segregated Witness，简称SW。用户在交易时，会把比特币传送到有别於传统的地址。当要使用这些比特币的时候，其签署 (即见证)并不会记录为交易ID的一部份，而是另外处理。也就是说，交易ID完全是由交易状态 (即结馀的进出) 决定，不受见证部份影响。

闪电网络（**Lightning Network**）：一个可扩展的微支付通道网络。交易双方若在区块链上预先设有支付通道，就可以多次、高频、双向地通过轧差方式实现瞬间确认的微支付；双方若无直接的点对点支付通道，只要网络中存在一条连通双方的、由多个支付通道构成的支付路径，闪电网络也可以利用这条支付路径实现资金在双方之间的可靠转移。

序列化：将一个数据结构转换成一个字节序列的过程。以太坊在内部使用的编码格式称为递归长度前缀编码（RLP）。

帕特里夏树：一种数据结构，它会存储每个帐户的状态。这个树的建立是通过从每个节点开始，然后将节点分成多达16个组，然后散列每个组，然后对散列结果继续散列，直到整个树有一个最后的“根散列”。该树具有重要的特性：（1）只有正好一个可能的树，因此，每个数据集对应一个可能的根散列（2）很容易的更新，添加，或者删除树节点，以及生成新的根散列，（3）不改变根散列的话没有办法修改树的任何部分，所以如果根散列被包括在签名的文档或有效区块中话，签名或工作证明可以担保整个树（ 4）任何人只可以提供一个下到特定节点的分支，可以加密得证明拥有确切内容的节点的确是在树里。帕特里夏树也被用来存储账户，交易已经叔块的内部存储。在这里能看到更详细的说明。

帐户随机数：每个账号的交易计数。这样可以防止重放攻击，其中一个交易发送比如 20个币从A到B，并可以被B重放一遍又一遍，直到不断抽干A的账户余额。

EVM代码：以太坊虚拟机代码，以太坊的区块链可以包含的编程语言的代码。与帐户相关联的EVM代码在每次消息被发到这个账户的时候被执行，并且具有读/写存储和自身发送消息的能力。

消息：一种由EVM代码从一个账户发送到另一个账户的“虚拟交易”。需要注意的是“交易”和“消息”在以太坊种是不同的;在以太坊用语的“交易”具体指的是物理的数字签名的一串数据，并且每个交易触发相关联的消息，但消息也可以通过EVM代码发送，在这种情况下，它们从不表示成任何数据。

储存：包含在每个帐户里的键／值数据库，其中键和值都是32个字节的字符串，但可以以其他方式包含任何东西。

外部拥有账户：通过私钥控制的账户。外部拥有账户不能包含EVM代码。

合约：一个包含并且受EVM的代码控制的账户。合约不能通过私钥直接进行控制，除非被编译成EVM代码，一旦合约被发行就没有所有者。

以太（**Ether**）：以太坊网络的内部基础的加密代币。以太是用来支付交易和以太坊交易的计算费用。

瓦斯：大致相当于计算步骤的计量。每一笔交易需要包括瓦斯的限制，还有愿意为每瓦斯支付的费用;矿工可以选择是否收录交易和收集费用。由包括原始消息以及任何可能被触发的子消息的交易产生的计算所使用的瓦斯总量，如果大于或者等于瓦斯的限制，则交易被处理。除非交易仍然有效并且费用仍然被矿工收集，否则瓦斯的总量小于限制则所有变更被还原。每一个操作都有瓦斯支出;对于大多数操作，花费是1瓦斯，尽管一些昂贵的操作会支出高达100瓦斯，交易本身会有500瓦斯的支出。

延伸概念

分散化应用程序：为了某些特定目的（如：在某些市场上连接买家和卖家，共享文件，网络文件存储，维持货币），无论是使用还是创建一个分散的网络，由许多人来运行的程序。基于以太坊的分散式的应用程序（也称为Ðapps，其中Ð为北欧字母“eth”）通常包括一个HTML/JavaScript的网页，并且如果在以太浏览器内部查看的话，可识别特殊的Javascript的API，用于发送交易数据到区块链，从区块链读取数据，和耳语，蜂群交互数据。一个Ðapp通常在区块链上有特定的相关合约，但有利于创造许多合约的Ðapps是完全可能的。

分散化组织（**GDO**）：一个没有中央领导，而是使用正式民主投票进程和共识主动性自我组织的结合来作为其基本操作原则。一个不太令人印象深刻，但有时混淆的概念是“地理上的分散化组织”（GDO），组织里人在相距甚远的地方工作，甚至可能都没有办公室; GDOs可能会有正式的中央领导。

忒修斯标准：用于查明一个组织的分散化程度的测试。测试如下：假设组织有N个人，然后外星人一次从组织中（比如每周一次）挑选K个人出来，摧毁他们存在，在每个群里以K个对组织不了解的新人来代替。现在为了让组织起作用，K可以高达多少人呢？在独裁政权里，当K＝1即独裁者被摧毁后就会失败。美国政府稍微好一点，但如果参议院和国会的所有638成员突然消失了的话，仍然会有很大的问题。但像比特币或BitTorrent即便对极高的K值也具有复

原性，因为新的代理人可以简单地根据自己的经济动机来填补缺失的角色。还有一个更严格的测试，拜占庭式修斯标准，它包含同一时间内随机的用恶意行为者取代K个用户一段时间，之后再替换成新用户。

委任式民主（或流动式民主）：一个对于DOs(分散式组织)和DAO(分散式自治组织)的治理机制，在默认情况下每个人对每件事情都投票，但在某些特定的问题上个人可以选择特定的他人为他们投票。这个想法概括了以下2种民主的权衡，完全直接民主(每个人都有相同的权力)和专家意见/有某些特定人提供的快速决策能力（允许人们自己顺从朋友，政治家，领域专家或者自己选择的任何人）。

部分被投机市场控制的理论上的政府：最初是由Robin Hanson提出的，为了管理政治组织治理机制。但它实际上是非常适用DOs和DAO的：通过预测市场来管理。从根本上，一些易于衡量成功的标准被选择，还有发行由成功标准的值来决定的代币，这些代币将在未来的某个时间（例如，1年后）被支付，对于每个可能要采取的行动都用一个这样的代币。这些代币都被兑换为相应的美元代币，如果相应的措施被执行,正好1美元会被支付 （如果相应的措施没有被执行，这两种类型的代币支付0美元，所以正在被执行的行动的概率不会影响价格）。市场预计的行动将有最好的结果，当其代币在市场上有高价格时会被执行。这提供了另一种自治的，选择机制，同时奖励专家的意见。

代币制度：本质上是可以交易的虚拟代替物。更正式地说，代币制度是一个数据库，它映射地址到数字，并具有以下属性，基本允许的操作是把N个代币从A转给B，条件是N是非负，且N不小于A的当前余额，授权该转账的证件由A进行数字签名。二次“发行”和“消费”的操作也可以存在，交易费用也可以被收集，许多当事人同时进行转账也是可能的。典型应用案例，包括货币，网络加密代币，公司的股份和数字礼品卡。

身份：一组可以加密验证的互动，具有同一个人创建的的属性。

唯一的身份：一组可以加密验证的互动，具有以下属性：同一个人创建的。再加上一个人不能有多个唯一身份的约束。

激励相容：如果每个人都更好的“遵守规则”而不是试图欺骗，除非至少要大量的人都同意同时一起欺骗，那么协议是激励相容的。

基本收入：每隔一段时间（比如几个月）就给每一个唯一的身份发送一定量的代币的想法。其最终目的是为了让不愿意工作或者不能工作的人能够依靠这份津贴活下来。这些代币可以简单的凭空制作出来，或者来自收益流（比如来自创收实体或政府）。为了单靠基本收入使人能够生活，可能会用到多个收益流的组合。

公益：一个为非常多的人提供了一个非常小的好处的服务。这样就没有任何个体对是否进行生产有影响力，因此也没有人有动力来支付。

声誉：身份的一个属性，其他实体认为这个身份可以（1）胜任一些特定的任务，或（2）在一些情况下是值得信赖。比如说不太可能因为短期的获利而出卖别人。

信任网络：如下的想法，如果A高度信任B，B高度信任C，则A可能是信任C的。为决定特定个体在特定概念下的可靠性的复杂而有力的机制，理论上可以由这个原则推断出来。

第三方托管：如果两个低信誉的实体所从事的贸易时，付款人可能希望把钱留在具有高信誉的第三方，并指示只有在产品交付后，才让第三方把钱发给收款人。这减少了付款人或收款人欺诈的风险。

保证金：放入合约里的涉及另外一方的数字资产，如果某些条件不满足时，该资产会自动被对方没收。

抵押：放入合约里的涉及另外一方的数字资产，如果某些条件不满足时，该资产会自动被销毁或据献给慈善或者基本收入基金。也许可以让利益广泛分配，但必须让特定的个人不能显著的受益。

参考

- 维基百科：http://en.wikipedia.org/wiki/Public-key_cryptography
- 《精通比特币》: https://github.com/imfly/bitcoinbook
- 巴比特论坛：http://8btc.com

JavaScript编码规范

主要依据下面的文档整理，局部有修改。

https://raw.githubusercontent.com/sivan/javascript-style-guide
https://github.com/airbnb/javascript JavaScript Standard Style

---

目录

类型

- 原始值: 存取直接作用于它自身。
    - `string`
    - `number`
    - `boolean`
    - `null`
    - `undefined`

```
var foo = 1;
var bar = foo;

bar = 9;

console.log(foo, bar); // => 1, 9
```

- 复杂类型: 存取时作用于它自身值的引用。
    - `object`
    - `array`
    - `function`

```
var foo = [1, 2];
var bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

⬆ 回到顶部

对象

- 使用直接量创建对象。

```
// bad
var item = new Object();

// good
var item = {};
```

- 不要使用保留字作为键名，它们在 IE8 下不工作。更多信息。

```
// bad
var superman = {
  default: { clark: 'kent' },
  privated: true
};

// good
var superman = {
  defaults: { clark: 'kent' },
  hidden: true
};
```

- 使用同义词替换需要使用的保留字。

```
// bad
var superman = {
  class: 'alien'
};

// bad
var superman = {
  klass: 'alien'
};

// good
var superman = {
  type: 'alien'
};
```

⬆ 回到顶部

数组

- 使用直接量创建数组。

```
// bad
var items = new Array();

// good
var items = [];
```

- 向数组增加元素时使用 Array#push 来替代直接赋值。

  ```javascript var someStack = [];
  ```

````
// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```
````

- 当你需要拷贝数组时，使用 Array#slice。jsPerf

```
var len = items.length;
var itemsCopy = [];
var i;

// bad
for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// good
itemsCopy = items.slice();
```

- 使用 Array#slice 将类数组对象转换成数组。

```
function trigger() {
  var args = Array.prototype.slice.call(arguments);
  ...
}
```

⬆ 回到顶部

字符串

- 使用单引号 `''` 包裹字符串。

```
// bad
var name = "Bob Parr";

// good
var name = 'Bob Parr';

// bad
var fullName = "Bob " + this.lastName;

// good
var fullName = 'Bob ' + this.lastName;
```
```

- 超过 100 个字符的字符串应该使用连接符写成多行。
- 注：若过度使用，通过连接符连接的长字符串可能会影响性能。jsPerf & 讨论.

```
// bad
var errorMessage = 'This is a super long error that was thrown because of Batman. When you stop to think about how Batman had anything to do with this, you would get nowhere fast.';

// bad
var errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// good
var errorMessage = 'This is a super long error that was thrown because ' +
  'of Batman. When you stop to think about how Batman had anything to do ' +
  'with this, you would get nowhere fast.';
```

- 程序化生成的字符串使用 Array#join 连接而不是使用连接符。尤其是 IE 下：jsPerf.

```
var items;
var messages;
var length;
var i;

messages = [{
  state: 'success',
  message: 'This one worked.'
}, {
  state: 'success',
  message: 'This one worked as well.'
}, {
  state: 'error',
  message: 'This one did not work.'
}];

length = messages.length;

// bad
function inbox(messages) {
  items = '<ul>';

  for (i = 0; i < length; i++) {
    items += '<li>' + messages[i].message + '</li>';
  }

  return items + '</ul>';
}

// good
function inbox(messages) {
  items = [];

  for (i = 0; i < length; i++) {
    // use direct assignment in this case because we're micro-optimizing.
    items[i] = '<li>' + messages[i].message + '</li>';
  }

  return '<ul>' + items.join('') + '</ul>';
}
```

⬆ 回到顶部

函数

- 函数表达式：

```
// 匿名函数表达式
var anonymous = function() {
  return true;
};

// 命名函数表达式
var named = function named() {
  return true;
};

// 立即调用的函数表达式（IIFE）
(function () {
  console.log('Welcome to the Internet. Please follow me.');
}());
```

- 永远不要在一个非函数代码块（if、while 等）中声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但它们的解析表现不一致。
- 注：ECMA-262 把 `块` 定义为一组语句。函数声明不是语句。阅读对 ECMA-262 这个问题的说明。

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
var test;
if (currentUser) {
  test = function test() {
    console.log('Yup.');
  };
}
```

- 永远不要把参数命名为 `arguments`。这将取代函数作用域内的 `arguments` 对象。

```
// bad
function nope(name, options, arguments) {
  // ...stuff...
}

// good
function yup(name, options, args) {
  // ...stuff...
}
```

⬆ 回到顶部

属性

- 使用 `.` 来访问对象的属性。

```
var luke = {
  jedi: true,
  age: 28
};

// bad
var isJedi = luke['jedi'];

// good
var isJedi = luke.jedi;
```

- 当通过变量访问属性时使用中括号 `[]` 。

```
var luke = {
  jedi: true,
  age: 28
};

function getProp(prop) {
  return luke[prop];
}

var isJedi = getProp('jedi');
```

⬆ 回到顶部

变量

- 总是使用 `var` 来声明变量。不这么做将导致产生全局变量。我们要避免污染全局命名空间。

```
// bad
superPower = new SuperPower();

// good
var superPower = new SuperPower();
```

- 使用 `var` 声明每一个变量。 这样做的好处是增加新变量将变的更加容易，而且你永远不用再担心调换错 `;` 跟 `,` 。

```
// bad
var items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// （跟上面的代码比较一下，看看哪里错了）
var items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

// good
var items = getItems();
var goSportsTeam = true;
var dragonball = 'z';
```

- 最后再声明未赋值的变量。当你需要引用前面的变量赋值时这将变的很有用。

```
// bad
var i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
var i;
var items = getItems();
var dragonball;
var goSportsTeam = true;
var len;

// good
var items = getItems();
var goSportsTeam = true;
var dragonball;
var length;
var i;
```

- 在作用域顶部声明变量。这将帮你避免变量声明提升相关的问题。

```
// bad
function () {
  test();
  console.log('doing stuff..');

  //..other stuff..

```

```
  var name = getName();

  if (name === 'test') {
    return false;
  }

  return name;
}

// good
function () {
  var name = getName();

  test();
  console.log('doing stuff..');

  //..other stuff..

  if (name === 'test') {
    return false;
  }

  return name;
}

// bad - 不必要的函数调用
function () {
  var name = getName();

  if (!arguments.length) {
    return false;
  }

  this.setFirstName(name);

  return true;
}

// good
function () {
  var name;

  if (!arguments.length) {
    return false;
  }

  name = getName();
  this.setFirstName(name);

  return true;
}
```

⬆ 回到顶部

提升

- 变量声明会提升至作用域顶部，但赋值不会。

```
// 我们知道这样不能正常工作（假设这里没有名为 notDefined 的全局变量）
function example() {
  console.log(notDefined); // => throws a ReferenceError
}

// 但由于变量声明提升的原因，在一个变量引用后再创建它的变量声明将可以正常工作。
// 注：变量赋值为 `true` 不会提升。
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// 解释器会把变量声明提升到作用域顶部，意味着我们的例子将被重写成：
function example() {
  var declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}
```

- 匿名函数表达式会提升它们的变量名，但不会提升函数的赋值。

```
function example() {
  console.log(anonymous); // => undefined

  anonymous(); // => TypeError anonymous is not a function

  var anonymous = function () {
    console.log('anonymous function expression');
  };
}
```

- 命名函数表达式会提升变量名，但不会提升函数名或函数体。

```
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

  var named = function superPower() {
    console.log('Flying');
  };
}

// 当函数名跟变量名一样时，表现也是如此。
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  var named = function named() {
    console.log('named');
  }
}
```

- 函数声明提升它们的名字和函数体。

```
function example() {
  superPower(); // => Flying

  function superPower() {
    console.log('Flying');
  }
}
```

- 了解更多信息在 JavaScript Scoping & Hoisting by Ben Cherry.

⬆ 回到顶部

比较运算符 & 等号

- 优先使用 `===` 和 `!==` 而不是 `==` 和 `!=`.
- 条件表达式例如 `if` 语句通过抽象方法 `ToBoolean` 强制计算它们的表达式并且总是遵守下面的规则：
    - 对象 被计算为 **true**
    - **Undefined** 被计算为 **false**
    - **Null** 被计算为 **false**

  - 布尔值 被计算为 布尔的值
  - 数字 如果是 **+0**、**-0** 或 **NaN** 被计算为 **false**，否则为 **true**
  - 字符串 如果是空字符串 `''` 被计算为 **false**，否则为 **true**

```
if ([0]) {
  // true
  // 一个数组就是一个对象，对象被计算为 true
}
```

- 使用快捷方式。

```
// bad
if (name !== '') {
  // ...stuff...
}

// good
if (name) {
  // ...stuff...
}

// bad
if (collection.length > 0) {
  // ...stuff...
}

// good
if (collection.length) {
  // ...stuff...
}
```

- 了解更多信息在 Truth Equality and JavaScript by Angus Croll.

⬆ 回到顶部

块

- 使用大括号包裹所有的多行代码块。

```
// bad
if (test)
  return false;

// good
if (test) return false;

// good
if (test) {
  return false;
}

// bad
function () { return false; }

// good
function () {
  return false;
}
```

- 如果通过 `if` 和 `else` 使用多行代码块，把 `else` 放在 `if` 代码块关闭括号的同一行。

```
// bad
if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}

// good
if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

⬆ 回到顶部

注释

- 使用 `/** ... */` 作为多行注释。包含描述、指定所有参数和返回值的类型和值。

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

  // ...stuff...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed in tag name
 *
 * @param {String} tag
 * @return {Element} element
 */
function make(tag) {

  // ...stuff...

  return element;
}
```

- 使用 `//` 作为单行注释。在评论对象上面另起一行使用单行注释。在注释前插入空行。

```
// bad
var active = true;  // is current tab

// good
// is current tab
var active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  var type = this._type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  var type = this._type || 'no type';

  return type;
}
```

- 给注释增加 `FIXME` 或 `TODO` 的前缀可以帮助其他开发者快速了解这是一个需要复查的问题，或是给需要实现的功能提供一个解决方式。这将有别于常见的注释，因为它们是可操作的。使用 `FIXME -- need to figure this out` 或者 `TODO -- need to implement` 。
- 使用 `// FIXME:` 标注问题。

```
function Calculator() {

  // FIXME: shouldn't use a global here
  total = 0;

  return this;
}
```

- 使用 `// TODO:` 标注问题的解决方式。

```
function Calculator() {

  // TODO: total should be configurable by an options param
  this.total = 0;

  return this;
}
```

⬆ 回到顶部

空白

- 使用 2 个空格作为缩进。

```
// bad
function () {
····var name;
}

// bad
function () {
·var name;
}

// good
function () {
··var name;
}
```

- 在大括号前放一个空格。

```
// bad
function test(){
  console.log('test');
}

// good
function test() {
  console.log('test');
}

// bad
dog.set('attr',{
  age: '1 year',
  breed: 'Bernese Mountain Dog'
});

// good
dog.set('attr', {
  age: '1 year',
  breed: 'Bernese Mountain Dog'
});
```

- 在控制语句（`if`、`while` 等）的小括号前放一个空格。在函数调用及声明中，不在函数的参数列表前加空格。

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

// good
function fight() {
  console.log('Swooosh!');
}
```

- 使用空格把运算符隔开。

```
// bad
var x=y+5;

// good
var x = y + 5;
```

- 在文件末尾插入一个空行。

```
// bad
(function (global) {
  // ...stuff...
})(this);
```

```
// bad
(function (global) {
  // ...stuff...
})(this);↵
↵
```

```
// good
(function (global) {
  // ...stuff...
})(this);↵
```

- 在使用长方法链时进行缩进。使用前面的点 `.` 强调这是方法调用而不是新语句。

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// bad
$('#items').
  find('.selected').
    highlight().
    end().
  find('.open').
    updateCount();

// good
$('#items')
  .find('.selected')
    .highlight()
    .end()
  .find('.open')
    .updateCount();

// bad
var leds = stage.selectAll('.led').data(data).enter().append('svg:svg').classed('l
ed', true)
    .attr('width', (radius + margin) * 2).append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin)
+ ')')
    .call(tron.led);

// good
var leds = stage.selectAll('.led')
    .data(data)
  .enter().append('svg:svg')
    .classed('led', true)
    .attr('width', (radius + margin) * 2)
  .append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin)
+ ')')
    .call(tron.led);
```

- 在块末和新语句前插入空行。

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo) {
  return bar;
}

return baz;

// bad
var obj = {
  foo: function () {
  },
  bar: function () {
  }
};
return obj;

// good
var obj = {
  foo: function () {
  },

  bar: function () {
  }
};

return obj;
```

⬆ 回到顶部

逗号

- 行首逗号: 不需要。

```
// bad
var story = [
    once
  , upon
  , aTime
];

// good
var story = [
  once,
  upon,
  aTime
];

// bad
var hero = {
    firstName: 'Bob'
  , lastName: 'Parr'
  , heroName: 'Mr. Incredible'
  , superPower: 'strength'
};

// good
var hero = {
  firstName: 'Bob',
  lastName: 'Parr',
  heroName: 'Mr. Incredible',
  superPower: 'strength'
};
```

- 额外的行末逗号：不需要。这样做会在 IE6/7 和 IE9 怪异模式下引起问题。同样，多余的逗号在某些 ES3 的实现里会增加数组的长度。在 ES5 中已经澄清了 (source)：

> Edition 5 clarifies the fact that a trailing comma at the end of an ArrayInitialiser does not add to the length of the array. This is not a semantic change from Edition 3 but some implementations may have previously misinterpreted this.

```
// bad
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn',
};

var heroes = [
  'Batman',
  'Superman',
];

// good
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn'
};

var heroes = [
  'Batman',
  'Superman'
];
```

⬆ 回到顶部

分号

- 使用分号。

```
// bad
(function () {
  var name = 'Skywalker'
  return name
})()

// good
(function () {
  var name = 'Skywalker';
  return name;
})();

// good (防止函数在两个 IIFE 合并时被当成一个参数
;(function () {
  var name = 'Skywalker';
  return name;
})();
```

了解更多.

⬆ 回到顶部

类型转换

- 在语句开始时执行类型转换。
- 字符串：

```
//  => this.reviewScore = 9;

// bad
var totalScore = this.reviewScore + '';

// good
var totalScore = '' + this.reviewScore;

// bad
var totalScore = '' + this.reviewScore + ' total score';

// good
var totalScore = this.reviewScore + ' total score';
```

- 使用 `parseInt` 转换数字时总是带上类型转换的基数。

```
var inputValue = '4';

// bad
var val = new Number(inputValue);

// bad
var val = +inputValue;

// bad
var val = inputValue >> 0;

// bad
var val = parseInt(inputValue);

// good
var val = Number(inputValue);

// good
var val = parseInt(inputValue, 10);
```

- 如果因为某些原因 `parseInt` 成为你所做的事的瓶颈而需要使用位操作解决性能问题时，留个注释说清楚原因和你的目的。

```
// good
/**
 * parseInt was the reason my code was slow.
 * Bitshifting the String to coerce it to a
 * Number made it a lot faster.
 */
var val = inputValue >> 0;
```

- 注：小心使用位操作运算符。数字会被当成 64 位值，但是位操作运算符总是返回 32 位的整数（source）。位操作处理大于 32 位的整数值时还会导致意料之外的行为。讨论。最大的 32 位整数是 2,147,483,647：

```
2147483647 >> 0 //=> 2147483647
2147483648 >> 0 //=> -2147483648
2147483649 >> 0 //=> -2147483647
```

- 布尔:

```
var age = 0;

// bad
var hasAge = new Boolean(age);

// good
var hasAge = Boolean(age);

// good
var hasAge = !!age;
```

⬆ 回到顶部

命名规则

- 避免单字母命名。命名应具备描述性。

```
// bad
function q() {
  // ...stuff...
}

// good
function query() {
  // ..stuff..
}
```

- 使用驼峰式命名对象、函数和实例。

```
// bad
var OBJEcttsssss = {};
var this_is_my_object = {};
var o = {};
function c() {}

// good
var thisIsMyObject = {};
function thisIsMyFunction() {}
```

- 使用帕斯卡式命名构造函数或类。

```
// bad
function user(options) {
  this.name = options.name;
}

var bad = new user({
  name: 'nope'
});

// good
function User(options) {
  this.name = options.name;
}

var good = new User({
  name: 'yup'
});
```

- 使用下划线 `_` 开头命名私有属性。

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';
```

- 使用 `_this` 保存 `this` 的引用。

```
// bad
function () {
  var self = this;
  return function () {
    console.log(self);
  };
}

// bad
function () {
  var that = this;
  return function () {
    console.log(that);
  };
}

// good
function () {
  var _this = this;
  return function () {
    console.log(_this);
  };
}
```

- 给函数命名。这在做堆栈轨迹时很有帮助。

```
// bad
var log = function (msg) {
  console.log(msg);
};

// good
var log = function log(msg) {
  console.log(msg);
};
```

- 注：IE8 及以下版本对命名函数表达式的处理有些怪异。了解更多信息到 http://kangax.github.io/nfe/。
- 如果你的文件导出一个类，你的文件名应该与类名完全相同。

```
// file contents
class CheckBox {
  // ...
}
module.exports = CheckBox;

// in some other file
// bad
var CheckBox = require('./checkBox');

// bad
var CheckBox = require('./check_box');

// good
var CheckBox = require('./CheckBox');
```

⬆ 回到顶部

存取器

- 属性的存取函数不是必须的。
- 如果你需要存取函数时使用 `getVal()` 和 `setVal('hello')` 。

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

- 如果属性是布尔值，使用 `isVal()` 或 `hasVal()` 。

```
// bad
if (!dragon.age()) {
  return false;
}

// good
if (!dragon.hasAge()) {
  return false;
}
```

- 创建 get() 和 set() 函数是可以的，但要保持一致。

```
function Jedi(options) {
  options || (options = {});
  var lightsaber = options.lightsaber || 'blue';
  this.set('lightsaber', lightsaber);
}

Jedi.prototype.set = function set(key, val) {
  this[key] = val;
};

Jedi.prototype.get = function get(key) {
  return this[key];
};
```

⬆ 回到顶部

构造函数

- 给对象原型分配方法，而不是使用一个新对象覆盖原型。覆盖原型将导致继承出现问题：重设原型将覆盖原有原型！

```
function Jedi() {
  console.log('new jedi');
}

// bad
Jedi.prototype = {
  fight: function fight() {
    console.log('fighting');
  },

  block: function block() {
    console.log('blocking');
  }
};

// good
Jedi.prototype.fight = function fight() {
  console.log('fighting');
};

Jedi.prototype.block = function block() {
  console.log('blocking');
};
```

- 方法可以返回 `this` 来实现方法链式使用。

```
// bad
Jedi.prototype.jump = function jump() {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function setHeight(height) {
  this.height = height;
};

var luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20); // => undefined

// good
Jedi.prototype.jump = function jump() {
  this.jumping = true;
  return this;
};

Jedi.prototype.setHeight = function setHeight(height) {
  this.height = height;
  return this;
};

var luke = new Jedi();

luke.jump()
  .setHeight(20);
```

- 写一个自定义的 `toString()` 方法是可以的，但是确保它可以正常工作且不会产生副作用。

```
function Jedi(options) {
  options || (options = {});
  this.name = options.name || 'no name';
}

Jedi.prototype.getName = function getName() {
  return this.name;
};

Jedi.prototype.toString = function toString() {
  return 'Jedi - ' + this.getName();
};
```

⬆ 回到顶部

事件

- 当给事件附加数据时（无论是 DOM 事件还是私有事件），传入一个哈希而不是原始值。这样可以让后面的贡献者增加更多数据到事件数据而无需找出并更新事件的每一个处理器。例如，不好的写法：

```
// bad
$(this).trigger('listingUpdated', listing.id);

...

$(this).on('listingUpdated', function (e, listingId) {
  // do something with listingId
});
```

更好的写法：

```
// good
$(this).trigger('listingUpdated', { listingId : listing.id });

...

$(this).on('listingUpdated', function (e, data) {
  // do something with data.listingId
});
```

⬆ 回到顶部

模块

- 模块应该以 `!` 开始。这样确保了当一个不好的模块忘记包含最后的分号时，在合并代码到生产环境后不会产生错误。详细说明
- 文件应该以驼峰式命名，并放在同名的文件夹里，且与导出的名字一致。
- 增加一个名为 `noConflict()` 的方法来设置导出的模块为前一个版本并返回它。
- 永远在模块顶部声明 `'use strict';` 。

```
// fancyInput/fancyInput.js

!function (global) {
  'use strict';

  var previousFancyInput = global.FancyInput;

  function FancyInput(options) {
    this.options = options || {};
  }

  FancyInput.noConflict = function noConflict() {
    global.FancyInput = previousFancyInput;
    return FancyInput;
  };

  global.FancyInput = FancyInput;
}(this);
```

⬆ 回到顶部

jQuery

- 使用 `$` 作为存储 jQuery 对象的变量名前缀。

```
// bad
var sidebar = $('.sidebar');

// good
var $sidebar = $('.sidebar');
```

- 缓存 jQuery 查询。

```
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...stuff...

  $('.sidebar').css({
    'background-color': 'pink'
  });
}

// good
function setSidebar() {
  var $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...stuff...

  $sidebar.css({
    'background-color': 'pink'
  });
}
```

- 对 DOM 查询使用层叠 `$('.sidebar ul')` 或 父元素 > 子元素 `$('.sidebar > ul')` 。 jsPerf
- 对有作用域的 jQuery 对象查询使用 `find` 。

```
// bad
$('ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();

// good
$sidebar.find('ul').hide();
```

⬆ 回到顶部

ECMAScript 5 兼容性

- 参考 Kangax 的 ES5 兼容表.

⬆ 回到顶部

测试

- **Yup.**

```
function () {
  return true;
}
```

⬆ 回到顶部

性能

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Loading...

⬆ 回到顶部

资源

推荐阅读

- Annotated ECMAScript 5.1

工具

- Code Style Linters
    - JSHint - Airbnb Style .jshintrc
    - JSCS - Airbnb Style Preset

其它风格指南

- Google JavaScript Style Guide

- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript
- JavaScript Standard Style

其它风格

- Naming this in nested functions - Christian Johansen
- Conditional Callbacks - Ross Allen
- Popular JavaScript Coding Conventions on Github - JeongHoon Byun
- Multiple var statements in JavaScript, not superfluous - Ben Alman

⬆ 回到顶部

ES5编码规范

本文译自 Airbnb JavaScript Style Guide

本文参考：https://github.com/yuche/javascript/blob/master/README.md

目 录

类型

- 1.1 基本类型: 直接存取基本类型。
  - `字符串`
  - `数值`
  - `布尔类型`
  - `null`
  - `undefined`

  ```
  const foo = 1;
  let bar = foo;

  bar = 9;

  console.log(foo, bar); // => 1, 9
  ```

- 1.2 复制类型: 通过引用的方式存取复杂类型。
  - `对象`
  - `数组`
  - `函数`

  ```
  const foo = [1, 2];
  const bar = foo;

  bar[0] = 9;

  console.log(foo[0], bar[0]); // => 9, 9
  ```

⬆ 返回目录

引用

- 2.1 对所有的引用使用 `const` ；不要使用 `var` 。

  > 为什么？这能确保你无法对引用重新赋值，也不会导致出现 bug 或难以理解。

  ```
  // bad
  var a = 1;
  var b = 2;

  // good
  const a = 1;
  const b = 2;
  ```

- 2.2 如果你一定需要可变动的引用，使用 `let` 代替 `var` 。

> 为什么？因为 `let` 是块级作用域，而 `var` 是函数作用域。

```
// bad
var count = 1;
if (true) {
  count += 1;
}

// good, use the let.
let count = 1;
if (true) {
  count += 1;
}
```

- 2.3 注意 `let` 和 `const` 都是块级作用域。

```
// const 和 let 只存在于它们被定义的区块内。
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

⬆ 返回目录

对象

- 3.1 使用字面值创建对象。

```
// bad
const item = new Object();

// good
const item = {};
```

- 3.2 如果你的代码在浏览器环境下执行，别使用 保留字 作为键值。这样的话在 IE8 不会运行。 更多信息。 但在 ES6 模块和服务器端中使用没有问题。

```
// bad
const superman = {
  default: { clark: 'kent' },
  privated: true,
};

// good
const superman = {
  defaults: { clark: 'kent' },
  hidden: true,
};
```

- 3.3 使用同义词替换需要使用的保留字。

```
// bad
const superman = {
  class: 'alien',
};

// bad
const superman = {
  klass: 'alien',
};

// good
const superman = {
  type: 'alien',
};
```

- 3.4 创建有动态属性名的对象时，使用可被计算的属性名称。

> 为什么？因为这样可以让你在一个地方定义所有的对象属性。

`` `javascript function getKey(k) { return `` a key named ${k}`; }

// bad const obj = { id: 5, name: 'San Francisco', }; obj[getKey('enabled')] = true;

// good const obj = { id: 5, name: 'San Francisco',

```
  [getKey('enabled')]: true,
};
```
```

- 3.5 使用对象方法的简写。
```

```
// bad
const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

- 3.6 使用对象属性值的简写。

> 为什么？因为这样更短更有描述性。

```
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

- 3.7 在对象属性声明前把简写的属性分组。

> 为什么？因为这样能清楚地看出哪些属性使用了简写。

```javascript
const anakinSkywalker = 'Anakin Skywalker';
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  episodeOne: 1,
  twoJedisWalkIntoACantina: 2,
  lukeSkywalker,
  episodeThree: 3,
  mayTheFourth: 4,
  anakinSkywalker,
};

// good
const obj = {
  lukeSkywalker,
  anakinSkywalker,
  episodeOne: 1,
  twoJedisWalkIntoACantina: 2,
  episodeThree: 3,
  mayTheFourth: 4,
};
```

⬆ 返回目录

数组

- 4.1 使用字面值创建数组。

```javascript
// bad
const items = new Array();

// good
const items = [];
```

- 4.2 向数组添加元素时使用 Arrary#push 替代直接赋值。

  ```javascript const someStack = [];

```
// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```
```

- 4.3 使用拓展运算符 `...` 复制数组。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

- 4.4 使用 Array#from 把一个类数组对象转换成数组。

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

⬆ 返回目录

解构

- 5.1 使用解构存取和使用多属性对象。

> 为什么？因为解构能减少临时引用属性。

```
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}

// good
function getFullName(obj) {
  const { firstName, lastName } = obj;
  return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

- 5.2 对数组使用解构赋值。

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

- 5.3 需要回传多个值时，使用对象解构，而不是数组解构。

> 为什么？增加属性或者改排序不会改变调用时的位置。

```
// bad
function processInput(input) {
  // then a miracle occurs
  return [left, right, top, bottom];
}

// 调用时需要考虑回调数据的顺序。
const [left, __, top] = processInput(input);

// good
function processInput(input) {
  // then a miracle occurs
  return { left, right, top, bottom };
}

// 调用时只选择需要的数据
const { left, right } = processInput(input);
```

⬆ 返回目录

Strings

- 6.1 字符串使用单引号 `''` 。

```
// bad
const name = "Capt. Janeway";

// good
const name = 'Capt. Janeway';
```

- 6.2 字符串超过 80 个字节应该使用字符串连接号换行。

- 6.3 注：过度使用字串连接符号可能会对性能造成影响。jsPerf 和 讨论.

```
// bad
const errorMessage = 'This is a super long error that was thrown because of Batman
. When you stop to think about how Batman had anything to do with this, you would
get nowhere fast.';

// bad
const errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// good
const errorMessage = 'This is a super long error that was thrown because ' +
  'of Batman. When you stop to think about how Batman had anything to do ' +
  'with this, you would get nowhere fast.';
```

- 6.4 程序化生成字符串时，使用模板字符串代替字符串连接。

> 为什么？模板字符串更为简洁，更具可读性。

```
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// bad
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

⬆ 返回目录

函数

- 7.1 使用函数声明代替函数表达式。

> 为什么？因为函数声明是可命名的，所以他们在调用栈中更容易被识别。此外，函数声明会把整个函数提升（hoisted），而函数表达式只会把函数的引用变量名提升。这条规则使得箭头函数可以取代函数表达式。

```
// bad
const foo = function () {
};

// good
function foo() {
}
```

- 7.2 函数表达式:

```
// 立即调用的函数表达式 (IIFE)
(() => {
  console.log('Welcome to the Internet. Please follow me.');
})();
```

- 7.3 永远不要在一个非函数代码块（ `if` 、 `while` 等）中声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但它们的解析表现不一致。
- 7.4 注意: ECMA-262 把 `block` 定义为一组语句。函数声明不是语句。阅读 ECMA-262 关于这个问题的说明。

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
let test;
if (currentUser) {
  test = () => {
    console.log('Yup.');
  };
}
```

- 7.5 永远不要把参数命名为 `arguments` 。这将取代原来函数作用域内的 `arguments` 对象。

```
// bad
function nope(name, options, arguments) {
  // ...stuff...
}

// good
function yup(name, options, args) {
  // ...stuff...
}
```

- 7.6 不要使用 `arguments` 。可以选择 rest 语法 `...` 替代。

> 为什么？使用 `...` 能明确你要传入的参数。另外 rest 参数是一个真正的数组，而 `arguments` 是一个类数组。

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

- 7.7 直接给函数的参数指定默认值，不要使用一个变化的函数参数。

```
// really bad
function handleThings(opts) {
  // 不！我们不应该改变函数参数。
  // 更加糟糕: 如果参数 opts 是 false 的话，它就会被设定为一个对象。
  // 但这样的写法会造成一些 Bugs。
  //（译注：例如当 opts 被赋值为空字符串，opts 仍然会被下一行代码设定为一个空对象。）
  opts = opts || {};
  // ...
}

// still bad
function handleThings(opts) {
  if (opts === void 0) {
    opts = {};
  }
  // ...
}

// good
function handleThings(opts = {}) {
  // ...
}
```

- 7.8 直接给函数参数赋值时需要避免副作用。

> 为什么？因为这样的写法让人感到很困惑。

```
var b = 1;
// bad
function count(a = b++) {
console.log(a);
}
count();  // 1
count();  // 2
count(3); // 3
count();  // 3
```

⬆ 返回目录

箭头函数

- 8.1 当你必须使用函数表达式（或传递一个匿名函数）时，使用箭头函数符号。

> 为什么?因为箭头函数创造了新的一个 `this` 执行环境（译注：参考 Arrow functions - JavaScript | MDN 和 ES6 arrow functions, syntax and lexical scoping），通常情况下都能满足你的需求，而且这样的写法更为简洁。
>
> 为什么不？如果你有一个相当复杂的函数，你或许可以把逻辑部分转移到一个函数声明上。

```
// bad
[1, 2, 3].map(function (x) {
  return x * x;
});

// good
[1, 2, 3].map((x) => {
  return x * x;
});
```

- 8.2 如果一个函数适合用一行写出并且只有一个参数，那就把花括号、圆括号和 `return` 都省略掉。如果不是，那就不要省略。

> 为什么？语法糖。在链式调用中可读性很高。
>
> 为什么不？当你打算回传一个对象的时候。

```
// good
[1, 2, 3].map(x => x * x);

// good
[1, 2, 3].reduce((total, n) => {
  return total + n;
}, 0);
```

⬆ 返回目录

构造器

- 9.1 总是使用 `class`。避免直接操作 `prototype`。

> 为什么? 因为 `class` 语法更为简洁更易读。

```javascript // bad function Queue(contents = []) { this._queue = [...contents]; } Queue.prototype.pop = function() { const value = this._queue[0]; this._queue.splice(0, 1); return value; }
```

```
// good
class Queue {
  constructor(contents = []) {
    this._queue = [...contents];
  }
  pop() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```
```

- 9.2 使用 `extends` 继承。

  > 为什么？因为 `extends` 是一个内建的原型继承方法并且不会破坏 `instanceof`。

  ```
  // bad
  const inherits = require('inherits');
  function PeekableQueue(contents) {
    Queue.apply(this, contents);
  }
  inherits(PeekableQueue, Queue);
  PeekableQueue.prototype.peek = function() {
    return this._queue[0];
  }

  // good
  class PeekableQueue extends Queue {
    peek() {
      return this._queue[0];
    }
  }
  ```

- 9.3 方法可以返回 `this` 来帮助链式调用。
  ```

```
// bad
Jedi.prototype.jump = function() {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function(height) {
  this.height = height;
};

const luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20); // => undefined

// good
class Jedi {
  jump() {
    this.jumping = true;
    return this;
  }

  setHeight(height) {
    this.height = height;
    return this;
  }
}

const luke = new Jedi();

luke.jump()
  .setHeight(20);
```

- 9.4 可以写一个自定义的 `toString()` 方法，但要确保它能正常运行并且不会引起副作用。

```
class Jedi {
  constructor(options = {}) {
    this.name = options.name || 'no name';
  }

  getName() {
    return this.name;
  }

  toString() {
    return `Jedi - ${this.getName()}`;
  }
}
```

⬆ 返回目录

模块

- 10.1 总是使用模组 (`import` / `export`) 而不是其他非标准模块系统。你可以编译为你喜欢的模块系统。

  > 为什么？模块就是未来，让我们开始迈向未来吧。

  ```
  // bad
  const AirbnbStyleGuide = require('./AirbnbStyleGuide');
  module.exports = AirbnbStyleGuide.es6;

  // ok
  import AirbnbStyleGuide from './AirbnbStyleGuide';
  export default AirbnbStyleGuide.es6;

  // best
  import { es6 } from './AirbnbStyleGuide';
  export default es6;
  ```

- 10.2 不要使用通配符 import。

  > 为什么？这样能确保你只有一个默认 export。

  ```
  // bad
  import * as AirbnbStyleGuide from './AirbnbStyleGuide';

  // good
  import AirbnbStyleGuide from './AirbnbStyleGuide';
  ```

- 10.3 不要从 import 中直接 export。

  > 为什么？虽然一行代码简洁明了，但让 import 和 export 各司其职让事情能保持一致。

  ```
  // bad
  // filename es6.js
  export { es6 as default } from './airbnbStyleGuide';

  // good
  // filename es6.js
  import { es6 } from './AirbnbStyleGuide';
  export default es6;
  ```

⬆ 返回目录

Iterators and Generators

- 11.1 不要使用 iterators。使用高阶函数例如 `map()` 和 `reduce()` 替代 `for-of` 。

> 为什么？这加强了我们不变的规则。处理纯函数的回调值更易读，这比它带来的副作用更重要。

```
const numbers = [1, 2, 3, 4, 5];

// bad
let sum = 0;
for (let num of numbers) {
  sum += num;
}

sum === 15;

// good
let sum = 0;
numbers.forEach((num) => sum += num);
sum === 15;

// best (use the functional force)
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;
```

- 11.2 现在还不要使用 generators。

> 为什么？因为它们现在还没法很好地编译到 ES5。 (译者注：目前(2016/03) Chrome 和 Node.js 的稳定版本都已支持 generators)

⬆ 返回目录

属性

- 12.1 使用 `.` 来访问对象的属性。

```
const luke = {
  jedi: true,
  age: 28,
};

// bad
const isJedi = luke['jedi'];

// good
const isJedi = luke.jedi;
```

- 12.2 当通过变量访问属性时使用中括号 `[]` 。

```
const luke = {
  jedi: true,
  age: 28,
};

function getProp(prop) {
  return luke[prop];
}

const isJedi = getProp('jedi');
```

⬆ 返回目录

变量

- 13.1 一直使用 `const` 来声明变量，如果不这样做就会产生全局变量。我们需要避免全局命名空间的污染。地球队长已经警告过我们了。（译注：全局，global 亦有全球的意思。地球队长的责任是保卫地球环境，所以他警告我们不要造成「全球」污染。）

```
// bad
superPower = new SuperPower();

// good
const superPower = new SuperPower();
```

- 13.2 使用 `const` 声明每一个变量。

> 为什么？增加新变量将变的更加容易，而且你永远不用再担心调换错 `;` 跟 `,` 。

```
// bad
const items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
const items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

- 13.3 将所有的 `const` 和 `let` 分组

> 为什么？当你需要把已赋值变量赋值给未赋值变量时非常有用。

```
// bad
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
let i;
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// good
const goSportsTeam = true;
const items = getItems();
let dragonball;
let i;
let length;
```

- 13.4 在你需要的地方给变量赋值，但请把它们放在一个合理的位置。

> 为什么？`let` 和 `const` 是块级作用域而不是函数作用域。

```
// good
function() {
  test();
  console.log('doing stuff..');

  //..other stuff..

  const name = getName();

  if (name === 'test') {
    return false;
  }

  return name;
}

// bad - unnecessary function call
function(hasName) {
  const name = getName();

  if (!hasName) {
    return false;
  }

  this.setFirstName(name);

  return true;
}

// good
function(hasName) {
  if (!hasName) {
    return false;
  }

  const name = getName();
  this.setFirstName(name);

  return true;
}
```

⬆ 返回目录

Hoisting

- 14.1 `var` 声明会被提升至该作用域的顶部，但它们赋值不会提升。`let` 和 `const` 被赋予了一种称为「暂时性死区（Temporal Dead Zones, TDZ）」的概念。这对于了解为什么 type of 不再安全相当重要。

```
// 我们知道这样运行不了
// （假设 notDefined 不是全局变量）
function example() {
  console.log(notDefined); // => throws a ReferenceError
}

// 由于变量提升的原因，
// 在引用变量后再声明变量是可以运行的。
// 注：变量的赋值 `true` 不会被提升。
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// 编译器会把函数声明提升到作用域的顶层，
// 这意味着我们的例子可以改写成这样：
function example() {
  let declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}

// 使用 const 和 let
function example() {
  console.log(declaredButNotAssigned); // => throws a ReferenceError
  console.log(typeof declaredButNotAssigned); // => throws a ReferenceError
  const declaredButNotAssigned = true;
}
```

- 14.2 匿名函数表达式的变量名会被提升，但函数内容并不会。

```
function example() {
  console.log(anonymous); // => undefined

  anonymous(); // => TypeError anonymous is not a function

  var anonymous = function() {
    console.log('anonymous function expression');
  };
}
```

- 14.3 命名的函数表达式的变量名会被提升，但函数名和函数函数内容并不会。

```javascript
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

  var named = function superPower() {
    console.log('Flying');
  };
}

// the same is true when the function name
// is the same as the variable name.
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  var named = function named() {
    console.log('named');
  }
}
```

- 14.4 函数声明的名称和函数体都会被提升。

```javascript
function example() {
  superPower(); // => Flying

  function superPower() {
    console.log('Flying');
  }
}
```

- 想了解更多信息，参考 Ben Cherry 的 JavaScript Scoping & Hoisting。

⬆ 返回目录

比较运算符 & 等号

- 15.1 优先使用 `===` 和 `!==` 而不是 `==` 和 `!=`.
- 15.2 条件表达式例如 `if` 语句通过抽象方法 `ToBoolean` 强制计算它们的表达式并且总是遵守下面的规则：
  - 对象 被计算为 **true**
  - **Undefined** 被计算为 **false**

  - **Null** 被计算为 **false**
  - 布尔值 被计算为 布尔的值
  - 数字 如果是 **+0**、**-0**、或 **NaN** 被计算为 **false**, 否则为 **true**
  - 字符串 如果是空字符串 `''` 被计算为 **false**，否则为 **true**

```
if ([0]) {
  // true
  // An array is an object, objects evaluate to true
}
```

- 15.3 使用简写。

```
// bad
if (name !== '') {
  // ...stuff...
}

// good
if (name) {
  // ...stuff...
}

// bad
if (collection.length > 0) {
  // ...stuff...
}

// good
if (collection.length) {
  // ...stuff...
}
```

- 15.4 想了解更多信息，参考 Angus Croll 的 Truth Equality and JavaScript。

⬆ 返回目录

代码块

- 16.1 使用大括号包裹所有的多行代码块。

```
// bad
if (test)
  return false;

// good
if (test) return false;

// good
if (test) {
  return false;
}

// bad
function() { return false; }

// good
function() {
  return false;
}
```

- 16.2 如果通过 `if` 和 `else` 使用多行代码块，把 `else` 放在 `if` 代码块关闭括号的同一行。

```
// bad
if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}

// good
if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

⬆ 返回目录

注释

- 17.1 使用 `/** ... */` 作为多行注释。包含描述、指定所有参数和返回值的类型和值。

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

  // ...stuff...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed in tag name
 *
 * @param {String} tag
 * @return {Element} element
 */
function make(tag) {

  // ...stuff...

  return element;
}
```

- 17.2 使用 `//` 作为单行注释。在评论对象上面另起一行使用单行注释。在注释前插入空行。

```
// bad
const active = true;  // is current tab

// good
// is current tab
const active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  const type = this._type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  const type = this._type || 'no type';

  return type;
}
```

- 17.3 给注释增加 `FIXME` 或 `TODO` 的前缀可以帮助其他开发者快速了解这是一个需要复查的问题，或是给需要实现的功能提供一个解决方式。这将有别于常见的注释，因为它们是可操作的。使用 `FIXME -- need to figure this out` 或者 `TODO -- need to implement` 。
- 17.4 使用 `// FIXME`：标注问题。

```
class Calculator {
  constructor() {
    // FIXME: shouldn't use a global here
    total = 0;
  }
}
```

- 17.5 使用 `// TODO`：标注问题的解决方式。

```
class Calculator {
  constructor() {
    // TODO: total should be configurable by an options param
    this.total = 0;
  }
}
```

⬆ 返回目录

空白

- 18.1 使用 2 个空格作为缩进。

```
// bad
function() {
····const name;
}

// bad
function() {
·const name;
}

// good
function() {
··const name;
}
```

- 18.2 在花括号前放一个空格。

```
// bad
function test(){
  console.log('test');
}

// good
function test() {
  console.log('test');
}

// bad
dog.set('attr',{
  age: '1 year',
  breed: 'Bernese Mountain Dog',
});

// good
dog.set('attr', {
  age: '1 year',
  breed: 'Bernese Mountain Dog',
});
```

- 18.3 在控制语句（ `if` 、 `while` 等）的小括号前放一个空格。在函数调用及声明中，不在函数的参数列表前加空格。

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

// good
function fight() {
  console.log('Swooosh!');
}
```

- 18.4 使用空格把运算符隔开。

```
// bad
const x=y+5;

// good
const x = y + 5;
```

- 18.5 在文件末尾插入一个空行。

```
// bad
(function(global) {
  // ...stuff...
})(this);
```

```
// bad
(function(global) {
  // ...stuff...
})(this);↵
↵
```

```
// good
(function(global) {
  // ...stuff...
})(this);↵
```

- 18.5 在使用长方法链时进行缩进。使用前面的点 `.` 强调这是方法调用而不是新语句。

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// bad
$('#items').
  find('.selected').
    highlight().
    end().
  find('.open').
    updateCount();

// good
$('#items')
  .find('.selected')
    .highlight()
    .end()
  .find('.open')
    .updateCount();

// bad
const leds = stage.selectAll('.led').data(data).enter().append('svg:svg').class('l
ed', true)
    .attr('width', (radius + margin) * 2).append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin)
+ ')')
    .call(tron.led);

// good
const leds = stage.selectAll('.led')
    .data(data)
  .enter().append('svg:svg')
    .classed('led', true)
    .attr('width', (radius + margin) * 2)
  .append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin)
+ ')')
    .call(tron.led);
```

- 18.6 在块末和新语句前插入空行。

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo) {
  return bar;
}

return baz;

// bad
const obj = {
  foo() {
  },
  bar() {
  },
};
return obj;

// good
const obj = {
  foo() {
  },

  bar() {
  },
};

return obj;
```

⬆ 返回目录

逗号

- 19.1 行首逗号：不需要。

```
// bad
const story = [
    once
  , upon
  , aTime
];

// good
const story = [
  once,
  upon,
  aTime,
];

// bad
const hero = {
    firstName: 'Ada'
  , lastName: 'Lovelace'
  , birthYear: 1815
  , superPower: 'computers'
};

// good
const hero = {
  firstName: 'Ada',
  lastName: 'Lovelace',
  birthYear: 1815,
  superPower: 'computers',
};
```

- 19.2 增加结尾的逗号: 需要。

> 为什么? 这会让 git diffs 更干净。另外，像 babel 这样的转译器会移除结尾多余的逗号，也就是说你不必担心老旧浏览器的尾逗号问题。

```
// bad - git diff without trailing comma
const hero = {
     firstName: 'Florence',
-    lastName: 'Nightingale'
+    lastName: 'Nightingale',
+    inventorOf: ['coxcomb graph', 'modern nursing']
}

// good - git diff with trailing comma
const hero = {
     firstName: 'Florence',
     lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing'],
}

// bad
const hero = {
  firstName: 'Dana',
  lastName: 'Scully'
};

const heroes = [
  'Batman',
  'Superman'
];

// good
const hero = {
  firstName: 'Dana',
  lastName: 'Scully',
};

const heroes = [
  'Batman',
  'Superman',
];
```

⬆ 返回目录

分号

- 20.1 使用分号

```
// bad
(function() {
  const name = 'Skywalker'
  return name
})()

// good
(() => {
  const name = 'Skywalker';
  return name;
})();

// good (防止函数在两个 IIFE 合并时被当成一个参数)
;(() => {
  const name = 'Skywalker';
  return name;
})();
```

Read more.

⬆ 返回目录

类型转换

- 21.1 在语句开始时执行类型转换。
- 21.2 字符串：

```
//  => this.reviewScore = 9;

// bad
const totalScore = this.reviewScore + '';

// good
const totalScore = String(this.reviewScore);
```

- 21.3 对数字使用 `parseInt` 转换，并带上类型转换的基数。

```
const inputValue = '4';

// bad
const val = new Number(inputValue);

// bad
const val = +inputValue;

// bad
const val = inputValue >> 0;

// bad
const val = parseInt(inputValue);

// good
const val = Number(inputValue);

// good
const val = parseInt(inputValue, 10);
```

- 21.4 如果因为某些原因 parseInt 成为你所做的事的瓶颈而需要使用位操作解决性能问题时，留个注释说清楚原因和你的目的。

```
// good
/**
 * 使用 parseInt 导致我的程序变慢，
 * 改成使用位操作转换数字快多了。
 */
const val = inputValue >> 0;
```

- 21.5 注: 小心使用位操作运算符。数字会被当成 64 位值，但是位操作运算符总是返回 32 位的整数（参考）。位操作处理大于 32 位的整数值时还会导致意料之外的行为。关于这个问题的讨论。最大的 32 位整数是 2,147,483,647：

```
2147483647 >> 0 //=> 2147483647
2147483648 >> 0 //=> -2147483648
2147483649 >> 0 //=> -2147483647
```

- 21.6 布尔:

```
const age = 0;

// bad
const hasAge = new Boolean(age);

// good
const hasAge = Boolean(age);

// good
const hasAge = !!age;
```

⬆ 返回目录

命名规则

- 22.1 避免单字母命名。命名应具备描述性。

```
// bad
function q() {
  // ...stuff...
}

// good
function query() {
  // ..stuff..
}
```

- 22.2 使用驼峰式命名对象、函数和实例。

```
// bad
const OBJEcttsssss = {};
const this_is_my_object = {};
function c() {}

// good
const thisIsMyObject = {};
function thisIsMyFunction() {}
```

- 22.3 使用帕斯卡式命名构造函数或类。

```
// bad
function user(options) {
  this.name = options.name;
}

const bad = new user({
  name: 'nope',
});

// good
class User {
  constructor(options) {
    this.name = options.name;
  }
}

const good = new User({
  name: 'yup',
});
```

- 22.4 使用下划线 `_` 开头命名私有属性。

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';
```

- 22.5 别保存 `this` 的引用。使用箭头函数或 Function#bind。

```
// bad
function foo() {
  const self = this;
  return function() {
    console.log(self);
  };
}

// bad
function foo() {
  const that = this;
  return function() {
    console.log(that);
  };
}

// good
function foo() {
  return () => {
    console.log(this);
  };
}
```

- 22.6 如果你的文件只输出一个类，那你的文件名必须和类名完全保持一致。

```
// file contents
class CheckBox {
  // ...
}
export default CheckBox;

// in some other file
// bad
import CheckBox from './checkBox';

// bad
import CheckBox from './check_box';

// good
import CheckBox from './CheckBox';
```

- 22.7 当你导出默认的函数时使用驼峰式命名。你的文件名必须和函数名完全保持一致。

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

- 22.8 当你导出单例、函数库、空对象时使用帕斯卡式命名。

```
const AirbnbStyleGuide = {
  es6: {
  }
};

export default AirbnbStyleGuide;
```

⬆ 返回目录

存取器

- 23.1 属性的存取函数不是必须的。
- 23.2 如果你需要存取函数时使用 `getVal()` 和 `setVal('hello')` 。

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

- 23.3 如果属性是布尔值，使用 `isVal()` 或 `hasVal()` 。

```
// bad
if (!dragon.age()) {
  return false;
}

// good
if (!dragon.hasAge()) {
  return false;
}
```

- 23.4 创建 `get()` 和 `set()` 函数是可以的，但要保持一致。

```
class Jedi {
  constructor(options = {}) {
    const lightsaber = options.lightsaber || 'blue';
    this.set('lightsaber', lightsaber);
  }

  set(key, val) {
    this[key] = val;
  }

  get(key) {
    return this[key];
  }
}
```

⬆ 返回目录

事件

- 24.1 当给事件附加数据时（无论是 DOM 事件还是私有事件），传入一个哈希而不是原始值。这样可以让后面的贡献者增加更多数据到事件数据而无需找出并更新事件的每一个处理器。例如，不好的写法：

```
// bad
$(this).trigger('listingUpdated', listing.id);

...

$(this).on('listingUpdated', function(e, listingId) {
  // do something with listingId
});
```

更好的写法：

```
// good
$(this).trigger('listingUpdated', { listingId : listing.id });

...

$(this).on('listingUpdated', function(e, data) {
  // do something with data.listingId
});
```

⬆ 返回目录

jQuery

- 25.1 使用 `$` 作为存储 jQuery 对象的变量名前缀。

```
// bad
const sidebar = $('.sidebar');

// good
const $sidebar = $('.sidebar');
```

- 25.2 缓存 jQuery 查询。

```
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...stuff...

  $('.sidebar').css({
    'background-color': 'pink'
  });
}

// good
function setSidebar() {
  const $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...stuff...

  $sidebar.css({
    'background-color': 'pink'
  });
}
```

- 25.3 对 DOM 查询使用层叠 `$('.sidebar ul')` 或 父元素 > 子元素 `$('.sidebar > ul')` 。 jsPerf
- 25.4 对有作用域的 jQuery 对象查询使用 `find` 。

```
// bad
$('ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();

// good
$sidebar.find('ul').hide();
```

⬆ 返回目录

ECMAScript 5 兼容性

- 26.1 参考 Kangax 的 ES5 兼容性.

⬆ 返回目录

ECMAScript 6 规范

- 27.1 以下是链接到 ES6 的各个特性的列表。

1. Arrow Functions
2. Classes
3. Object Shorthand
4. Object Concise
5. Object Computed Properties
6. Template Strings
7. Destructuring
8. Default Parameters
9. Rest
10. Array Spreads
11. Let and Const
12. Iterators and Generators
13. Modules

⬆ 返回目录

测试

- 28.1 **Yup.**

```
function() {
  return true;
}
```

⬆ 返回目录

性能

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Loading...

⬆ 返回目录

资源

- EC6保留字（Reserved keywords as of ECMAScript 6）
- ECMAScript的严格模式(The Strict Mode of ECMAScript).aspx)

⬆ 返回目录

关于亿书

更新细内容，请访问亿书官网： http://ebookchain.org

或查阅亿书白皮书： http://ebookchain.org/ebookchain.pdf

名字解释

亿书，是一款协作创作软件，也是一个自出版平台，更是一个出版交易市场。其名字，是Ebook的直译，是产品的统称。在通常情况下，它以一个去中心化的软件出现，就像比特币的客户端。

主要特点

最终的版本，将具备如下特点：

- 用户可以用来撰写博客文章，轻松将博客整理成电子书，并一键发布;
- 用户能很简单地绑定自己的域名，定制个性页面，供全世界浏览，轻松搭建自出版平台，取代市场上的各类博客;
- 用户能很方便与他人合作，类似github的代码托管可以多分支处理，后台给出多维度统计，方便利益分配;
- 产品背后，由加密货币亿书币（Ebookcoin)驱动，为文章、电子书、视频、图片自动追加版权保护和认证，保护用户权益;
- 基于区块链，提供文章、多媒体、电子书的透明交易，收益分配管理等;
- 为传统的出版机构、媒体等，提供侧链功能，帮助他们实现基于区块链的版权保护和交易;
- 优化开发接口，方便传统博客、论坛等知识分享软件集成使用，以更加宽容开放的心态，吸引第三方开发者，鼓励世界知识分享。

关联产品

- 亿书链(Ebookchain)：这是亿书官方平台http://ebookchain.org，该平台将聚合亿书的全部资源，包括亿书博客(http://blog.ebookchain.org)，亿书论坛(http://forum.ebookchain.org)等等。

该平台也是亿书的重要实验节点，逐步使用亿书打造成亿书官网的自出版平台。同时，聚合亿书网络信息（比如：用户发布的成品电子书等），提供链外用户的查询、使用和推广。

- 亿书币（Ebookcoin)：使用Node.js开发的加密货币产品，是亿书的底层技术支撑。我们会着重从易用性角度去开发设计，将其开发成Node.js的普通组件，方便开发者安装、使用和集成。

主要创新

亿书是目前

- 世界上第一个针对普通老百姓的区块链产品;
- 世界上第一个边使用边打造的软件产品，能用、好用、易用是其根本目的;（为了方便的写好这本书，已经相继贡献社区多个开源产品，亿书将是最完美的一个）
- 世界上第一个去中心化的博客产品;
- 世界上第一个主动去适配和兼容已有产品的区块链产品;（以太坊、Lisk都是要吸引其他开发者加入，亿书将主动开发各类插件去集成市面上各类著名软件产品，如：ghost博客、NodeBB论坛等）
- 世界上第一个提供详细培训教程（本书）的软件产品;
- 世界上第一个培养专业技术团队，为第三方提供专业技术服务的区块链产品。

亿书资源

本书地址： https://github.com/imfly/bitcoin-on-nodejs

亿书白皮书： http://ebookchain.org/ebookchain.pdf

亿书官网： http://ebookchain.org

亿书官方QQ群：185046161（亿书完全开源开放，欢迎各界小伙伴参与）

总之，亿书，是一本充满魔幻与期待、思考与实践的书，是把比特币的智慧继承与延伸的一款产品，是来源于社区并贡献于社区的产物！

简介

区块链俱乐部，英文名ChainClub，简称CC，目标是围绕区块链研发、运营与投资，不断发现、培养和聚合区款链从业人才，打造国内最具影响力的区块链管理者、开发者、运营者分享与交流平台。

官方网站： http://chainclub.org

运作模式

ChainClub，实行会员免费申请、邀请推荐、实名认证的加入机制。通过定期组织区块链茶座等线下活动，通过官网、公众号等媒介，提供业内资讯、技术分享、创业故事等信息，为会员打造一套成熟的集线上线下讨论和分享于一体的社交平台。

会员构成

ChainClub，成员遍布全世界，如果您是从事区块链开发、研究和教育等行业的各类人才，包括但不限于下述之一的，都可以加入我们。

- IT公司和各行业企业研发部门的CTO、技术副总裁、首席架构师、技术总监、项目总监、产品总监、运维总监等技术管理者;
- 行政管理部门CEO、CFO、法律顾问、营销总监等运营管理人员;
- 各大院校、研究院所的高级教师、研究员、学生等

会员权益

在这里，您可以获得：

交流分享: 分享区款链研究心得，探讨开发经验、管理实践等; 拓展人脉: 结识更多区款链业界领袖，与内行专家零距离交流，拓展人脉关系; 树立品牌: 增加公司曝光度，打造企业技术品牌，提升个人影响力; 学习成长: 洞悉区款链行业产品和技术变革，掌握最新技术和发展趋势; 职业生涯: 企业可以快速找到合适的人才，人才有更多方便直接的择业机会; 亿书服务: 免费获得亿书开源社区的有关技术、培训和咨询服务，获得亿书基金的技术孵化机会，助您成就一番事业

加入条件

-身心健康，文明礼貌，对新事物、新技术、新观点有好奇心，尊重不同观点和思想; -尊重亿书开源理念和免费分享的价值观，愿意为开源社区贡献自己的智慧和力量； -热爱区款链事业，具备加密货币研发、运营、投资等某一方面的从业经验者优先; -懂软硬件开发技术者优先，熟悉C/C++,java,javascript,ruby,python,go等一种或几种开发语言更佳; -懂互联网、软件产品运营管理者优先，有大公司运营管理经验更佳; -有写作经验者优先，懂一门或多门外语能力更佳; -积极向亿书反馈亿书社区和各类产品使用体验，帮助开源社区发展。

加入步骤

严格采取实名制，装酷、摆谱、自以为聪明，不按照要求执行的，一律拒绝！我们会严格保护会员隐私，绝不会用于俱乐部各类活动之外的其他用途。

1.请加我微信（kubying），注明：“真实姓名 + 公司 + 职务，申请加入俱乐部”，会拉你到俱乐部微信群。

2.请将您的真实姓名 + 公司 + 职务，擅长的领域，常用联系方式（电话、QQ、微信、skype等），如果是技术人员，还应包括github等代码托管网站帐号等，发送邮件到: support@chainclub.org 方便活动联系。

3.关注中国区块链俱乐部官方网站，网址：http://chainclub.org，了解往期活动内容和最新活动状态。

4.关注区块链俱乐部公众号（chainclub），接受最新消息推送。

联系方式

俱乐部：http://chainclub.org

官方邮箱： support@chainclub.org

亿书QQ交流群： 185046161

亿书官网： http://ebookchain.org

亿书团队 于2016年5月1日

关于作者

lmfly，

微信：**kubying**

一个4岁孩子的父亲。把快乐作为一切取舍的唯一考量。感念生命短促，顾惜时间胜于金钱和虚名。希望通过技术结识天下朋友，做更多自己喜欢、擅长、有价值的事情。喜欢新挑战和新事物，较早进入加密货币领域，开发、投资、管理至今。

10+年开发经验，全栈开发工程师，精通java、ruby on rails、node.js，是ruby on rails和node.js早期开发实践者。在大型国企从事管理工作多年，在企业级、物联网领域有大项目开发、管理、实践经验，擅长Web开发、数据挖掘、即时应用产品设计和系统架构。

后记

“无论多么伟大的事情，背后往往都是人人都明白的简单方法。”这是我在最近转发一篇朋友圈文章的时候写下的。这篇文章介绍的是加拿大一位年轻人，斯科特▪杨（Scott Young），如何高效学习的事情。他通常花费比别人少很多的时间，却获得很好的学习成绩。他的法宝其实非常简单，确定目标之后，遵循“学习-反馈-改进”这样的循环，直到目标达成。实际上，不管你信不信，任何事情只要成功了，都是有意或无意的践行了这一方法。在编程开发领域，用一个比较流行的词就叫“增量开发”，再简单一点就是“迭代”。

世界上，没有一下子就能做好、做成的事情，“每一件与众不同的绝世好东西，其实都是以无比寂寞的勤奋为前提的，要么是血，要么是汗，要么是大把大把的曼妙青春好时光”，这些付出就是用在反复的打磨、反复的改进、反复的迭代之中。而现实往往是，历经这样的磨砺，东西不好都难，或者说不惊世骇俗就是低估了它，根本不需要去关心经过这样的学习之后，能不能找到工作，更不用担心这样做出来的东西，有没有人认可。相反，仅仅为了工作，为了别人的眼光，反而没有了坚持做下去的动力，最终的结果，往往是蹉跎光阴，浪费生命。

我无意间又一次践行了这样的做法。最初，撰写和分享书里这些文章的时候，目的非常单纯，只是想通过代码了解区块链背后的秘密，没有想过要教育别人，当然更没有想到会结集出版。所以，本书是在完全开源开放的条件下完成的，每一章就像编程一样通过代码库管理并提交到github上，定稿之后发布到相关论坛和社区，基本上每周一篇。写作时，我会反复修改；发布到论坛之后，根据反馈再修改；写到后面的章节，感觉前面有问题，还会修改。书，虽然只有30篇左右，但是版本库的提交记录接近260次（这是完整可见的修改次数）。

尽管如此，这本书依然不够完美，或者用粗制滥造形容更为合适，毕竟短短的几个月时间，又怎么可能尽数区块链的奇妙和全部细节。文章不可避免会存在这样那样的问题， 至少在内容上，就存在一个重大缺陷，比如：区块链发展到现在，单一主链已经显现出明显的劣势，侧链注定是它的未来，但是这本书里没有涉及。不是不想，而是就区块链目前的现状和我本人的能力，还不足以把这个问题描述清楚，需要讨论和研究的问题太多。这个问题，我当然会坚持研究下去，并把它体现在亿书产品里。虽然不能在这本书里与大家见面，但是大家可以浏览亿书源码，探究它的实现。待代码成熟，我会继续分享，补全这部分内容。

在历经8个多月，以及很多个挑灯夜战的辛苦之后，原本怀疑是否会中途放弃的写作分享，终于暂告一段落，内心多少还是有点澎湃的。澎湃的不是终于完整分享了全部内容，而是担心这些内容能否给您带来些什么，能否为中国的区块链发展做出点什么。无论好与坏，它就在这里。另外，本书虽已完结，但分享仍将继续，因为区块链的发展才刚刚开始。

是为记！

2016.10.10